VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulátor Petriho sítí

Petri net simulator

2014 Bc. Jakub Čech

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Čech**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Simulátor Petriho sítí
Petri Net Simulator

Zásady pro vypracování:

Cílem práce je vytvořit simulátor Petriho sítí na platformě Eclipse RCP, který bude umožňovat simulaci Petriho sítě. Zadání práce navazuje na existující projekt Editor Petriho sítí pro Eclipse RCP.
Simulátor by měl umožňovat:
1. Simulaci Petriho sítě.
2. Náhodnou volbu zvolených parametrů přechodů.
3. V síti se při simulaci bude pracovat s tokeny, které budou reprezentovány jako objekty.
4. Jednotlivé tokeny-objekty budou obsahovat uživatelské atributy, které budou ovlivňovat simulaci.
5. Pro každý přechodu bude možno nadefinovat, zda dochází k vytváření nových tokenů, nebo pouze modifikaci stávajících. Modifikace bude definována skriptovacím jazykem, přičemž bude možno měnit nebo nastavovat jednotlivé parametry tokenů.
6. Výsledky a průběh simulace bude zaznamenáván a výsledky se zobrazí ve vhodném grafu.
7. Simulaci bude možno spouštět opakovaně s možností vyhodnocení výsledků pomocí statistických metod.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] DARWIN, Ian F. Java cookbook. 2nd ed. Sebastopol, CA: O'Reilly, c2004, xxiv, 829 p. ISBN 05-960-0701-9. Dostupné z: http://it-ebooks.info/book/2249/
[3] Eclipse [online]. 2011 [cit. 2011-06-04]. Dostupné z WWW: <http://www.eclipse.org/>.

Dále podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst.9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB – TU Ostrava.*

V Ostravě 2014 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 2014 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, především mému vedoucímu práce, panu Ing. Davidu Ježkovi, Ph.D.. Bez jejich nezištné pomoci by tato práce nevznikla.

Abstrakt

Tato práce popisuje vytvoření simulace Petriho sítě do editoru, který je výsledkem předešlé diplomové práce [1]. Dále je řešeno provádění skriptů za běhu aplikace, úprava GMF aplikace a vytvoření grafů pomocí JFreeChart.

Klíčová slova: Petriho síť, diplomová práce, Java, Eclipse, GMF, EMF, Groovy, Skriptování, JFreeChart

Abstract

This thesis is about creating simulation of Petri net into editor from thesis [1]. Next will be describe, how use script in running application, how edit GMF and how to make Chart with JFreeChart.

Keywords: Petri net, diploma thesis, Java, Eclipse, GMF, EMF, Groovy, Scripting, JFreeChart

Seznam použitých zkratek a symbolů

GMF	–	Graphical Modeling Project
EMF	–	Eclipse Modeling Framework
GEF	–	Graphical Editing Framework
API	–	Application Programming Interface
JVM	–	Java Virtual Machine
JSR	–	Java Specification Request
API	–	Application Programming Interface
SVN	–	Subversion

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce popisuje postup řešení problematiky na téma Simulátor Petriho sítí. V následujících kapitolách bude popisován postupný vývoj řešení.

Nejprve začneme seznámením s problematikou Petriho sítí. Následně si popíšeme technologie použité během práce. Další kapitoly se již věnují samotnému řešení, jak práce vzniká od prvního spuštění a prvních pokusů o vytvoření simulace přes hlubší proniknutí do problematiky GMF aplikací, až k vytvoření fungující simulace dle zadání.

Nejdříve bude probráno doplnění maličkostí do editoru a začátky programování základní simulace. V závěru této kapitoly je zjištěno, že práci není možné v dodaném editoru vytvořit a z toho plyne nutnost úpravy značné části tohoto editoru. Jedná se o problémy vzniklé již během tvorby dodaného editoru, které značně ovlivnily průběh této práce. Tomu je věnována další kapitola, která popisuje postupnou změnu celého modelu, rozpohybování simulace ve vykreslené síti a další důležité úpravy nutné pro splnění několika bodů zadání.

V závěru práce je probráno dokončení simulace o přidání další funkčnosti, kterou je použití skriptovacího enginu vyhodnocujícího skripty během samotné simulace, vytvoření akcí nutných pro opakované spouštění a doplnění časového aspektu do simulace. Po dokončení simulace jsou nakonec přidány záznamy běhu, které řeší statistiku provádění sítě a graf zobrazující historii použití jednotlivých objektů používaných v síti.

Nakonec je zhodnocen konečný stav řešení a jsou popsány základní nedostatky, které však nejsou předmětem této práce.

2 O Petriho sítích

V této kapitole vycházím z [2, 3, 4].

2.1 Seznámení s Petriho sítěmi

Petriho sítě jsou formálně a grafický příjemný jazyk sloužící k modelování procesů. Petriho sítě byly vyvíjeny od počátku 60. let 20. století, kdy prof. Carl Adam Petri definoval jejich jazyk ve své disertační práci. Bylo to poprvé, kdy byla formulována teorie diskrétních paralelních systémů. Jejich jazyk je rozšířením teorie automatů tak, že nám umožňuje vyjádřit současně se vyskytující události.

2.2 Historie Petriho sítí

Petriho sítě byly původně vyvinuty v 60. a 70. letech 20. století a byly shledány jako jeden z nejvhodnějších způsobů pro popsání a analýzu synchronizace, komunikace a sdílení zdrojů mezi souběžnými procesy. Nicméně pokusy o praktické využití Petriho sítí odhalily dva vážné nedostatky.

1. **absence datových typů**, což způsobovalo, že modely se často stávaly příliš velkými, protože veškerá manipulace s daty musela být vyjádřena pouze pomoci míst a přechodů.

2. **absence hierarchizace**, a proto nebylo možné vytvářet velké modely pomoci samostatných subprocesů s definovaným vstupním a výstupním rozhraním.

Další vývoj high-level Petriho sítí v 70. a 80. tyto dva nedostatky odstranil. Vzniklé barevné Petriho sítě jsou jeden z nejznámějších dialektů high-level Petriho sítí. Barevné Petriho sítě již obsahují datové typy i hierarchický rozklad zachovávající kvality původních Petriho sítí.

Petriho sítě byly postupně obohacovány a zobecňovány tak, aby jejich modelovací schopnost vyhovovala praktickým potřebám. Z toho důvodu časem vznikly následující typy.

- **C/E (Condition/Event) Petriho sítě** - základní rozšíření konečného automatu vytvořené C.A.Petrim. Model se skládá z událostí (events) a podmínek (conditions), které musí být splněny, aby mohla nastat určitá událost. Vazby mezi událostmi a podmínkami jsou znázorněny pomocí orientovaných hran. Každá podmínka může obsahovat pouze jediný token, udávající že podmínka je splněna.

- **P/T (Place/Transitions) Petriho sítě** - rozšíření původního C/E konceptu. Podmínky byly nahrazeny místy a místo událostí se používají přechody. Oproti původnímu konceptu je zde povoleno uchovávat v místě více než jeden token. Dále jsou nově zavedeny váhy (násobnosti) jednotlivých hran, které udávají, kolik tokenů hrana přesouvá. Rovněž je možné omezit kapacitu míst.

- **P/T Petriho sítě s inhibičními hranami** - ke konceptu P/T síti je přidána inhibiční hrana. Jedná se o speciální hranu, jež může směřovat pouze od místa k přechodu a místo šipky se používá kolečko. Tato hrana říká, že přechod je proveditelný jen tehdy, je-li v místě, z něhož inhibiční hrana vystupuje, méně tokenů, nežli je násobnost této hrany.
- **P/T Petriho sítě s prioritami** - rozšíření C/E konceptu, kdy ke každému přechodu je přiřazeno celé nezáporné číslo udávající prioritu přechodu. Při provádění se provádí přechod proveditelný přechod s nejvyšší prioritou.
- **TPN Časované (Timed)Petriho sítě** - oproti všem dříve zmíněným přidává k P/T síti časový element. To nám umožňuje zachytit, že provedení přechodu trvá nějakou dobu a přibližuje tak například simulaci procesu výroby reálnému stavu, když výroba různých produktů trvá různou dobu.
- **CPN Barevné (Coloured, barvené) Petriho sítě** - obyčejná P/T síť, v níž je možno pracovat s různými typy tokenů. Pak neukládá množinu obecných tokenů, ale multimnožinu. Výhodou multimnožiny je, že jeden prvek v ní může být obsažen vícekrát. Dále umožňuje přiřadit přechodu vstupní podmínku, jež upravuje jeho proveditelnost na základě hodnot vstupních tokenů.
- **HPN Hierarchické (Hierarchical) Petriho sítě** - umožňuje obyčejné P/T síti za místa a přechody substituovat menší sítě s definovaným vstupním a výstupním rozhraním.
- **OOPN Objektově orientované (Object Oriented) Petriho sítě** - každý token představuje instanci určité třídy s vlastními atributy.

Cílem této práce je vytvořit program, který bude umožňovat simulaci objektově orientované, časované Petriho sítě s možností vytvoření hierarchické struktury.

2.3 Popis a stavová analýza P/T Petriho sítě

Pro pochopení základní problematiky si vysvětleme základní P/T Petriho síť. Tyto sítě bývají nazývány také černobílé sítě. Diagram P/T Petriho sítě obsahuje následujícími objekty:

- ***místa*** (places) zobrazovanými kroužky,
- ***přechody*** (transitions) zobrazovanými obdélníky (případně úsečkami),
- orientované ***hrany*** reprezentované šipkami vedoucími od míst k přechodům, nebo šipkami od přechodů k místům,
- ***tokeny*** (zobrazenými tečkami v kroužcích míst) jednotlivé objekty mapy,
- ***kapacity*** (zobrazené čísly u míst) udávají maximální počet tokenů, který se může v místě nacházet,

- ***váhy*** (zobrazené číslem u hrany) udávající počet tokenů, který hrana přesouvá,
- ***počáteční značení*** udává počty tokenů v jednotlivých místech.

Kapacity míst a násobnosti hran nejsou povinné. Při nevyplnění kapacity se předpokládá, že kapacita místa je nekonečná a hrana bez zadané násobnosti má násobnost 1.

Nyní si definujme černobílou Petriho síť formálně. První si musíme vytvořit strukturu definovanou definicí 2.1.

Definice 2.1 *(převzato z [2])*
Struktura Petriho sítě *(PN-struktura) je pětici* $\langle P, T, I, O, H \rangle$*, kde* $T \bigcap P = \emptyset$ *a*

- P *- konečná množina míst,*
- T *- konečná množina přechodů,*
- I, O, H *- zobrazení typu* $T \longrightarrow P_{MS}$*, po řadě tzv. vstupní, výstupní a vstupní inhibiční funkce*[1].

Tato struktura nám zachycuje statický diagram sítě. V celé této práci bude řešen specifický případ kdy $H = \emptyset$. To znamená že program neumožňuje pracovat s inhibičními hranami. Proto tyto hrany budou v další teorii vypuštěny.

Nyní již jsme schopni popsat diagramem Petriho sítě určitý proces, ale zatím nemůžeme popsat stav, ve kterém se proces nachází. Abychom mohli popsat stav Petriho sítě, musíme ke struktuře sítě definovat značení.

Definice 2.2 *(převzato z [3])*
Značení (Marking) M ***Petriho sítě*** $PN = \langle P, T, I, O \rangle$ *je zobrazení z množiny míst* P *do množiny nezáporných celých čísel* $\mathbb{N}_0$

$$M : P \longrightarrow \mathbb{N}_0.$$

Když spojíme značení a strukturu sítě, získáme systém Petriho sítě.

Definice 2.3 *(převzato z [2])*
Systém Petriho sítě *(PN-systém) je pětice* $\langle P, T, I, O, M_0 \rangle$*, kde*

- $\langle P, T, I, O \rangle$ *je struktura Petriho sítě (viz definice 2.1)*[2]*,*
- M_0 *je zobrazení* $P \longrightarrow \mathbb{N}_0$*, které udává počáteční značení. To znamená značení (viz definice 2.2) před provedením (viz definice 2.4) jakéhokoli přechodu.*

[1] P_{MS} - je množina všech multimnožin nad množinou P
[2] jelikož $H = \emptyset$, pak toto značení vypadlo

K následujícím definicím bude potřebovat formální značení, které si nejprve zadefinujeme a popíšeme.

- $I(t)$ - multimnožina vstupních míst přechodu t,
- $O(t)$ - multimnožina výstupních míst přechodu t,
- $I(t,p)$ - násobnost prvku p v multimnožině $I(t)$, násobnost hrany z místa p do přechodu t,
- $O(t,p)$ - násobnost prvku p v multimnožině $O(t)$, násobnost hrany z přechodu t do místa p,
- ${}^\bullet t = \{p \in P : I(t,p) > 0\}$ - množina vstupních míst přechodu t,
- $t^\bullet = \{p \in P : O(t,p) > 0\}$ - množina výstupních míst přechodu t,
- ${}^\bullet p = \{t \in T : O(t,p) > 0\}$ - množina vstupních přechodů místa p,
- $p^\bullet = \{t \in T : I(t,p) > 0\}$ - množina výstupních přechodů místa p,
- $E(M)$ - množina proveditelných přechodů při značení M,
- $M \rightarrow^t M'$ - stav sítě se mění z M na M' v důsledku provedení přechodu t.

S definovaným systémem se můžeme pustit do definování již zmíněného provedení přechodu.

Definice 2.4 *(převzato z [2])*
Provedení (odpálení, fire) přechodu

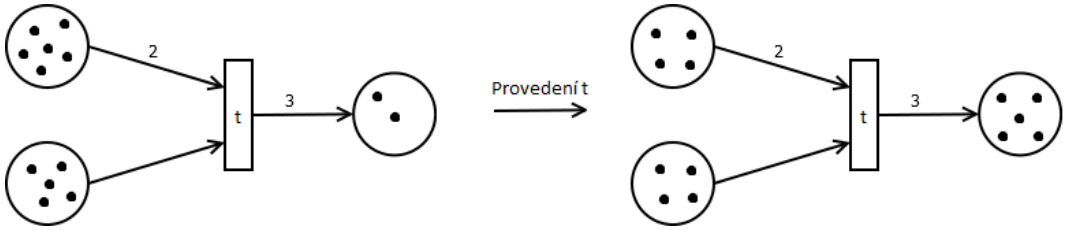
$$(\forall p \in P)[M'(p) = M(p) + O(t,p) - I(t,p)].$$

K tomu, aby bylo možné při aktuálním značení M přechod provést, musí být takový přechod proveditelný.

Definice 2.5 *(převzato z [2])*
Přechod je při značení M ***proveditelný (Enable)****, platí-li*

$$(\forall p \in {}^\bullet t)[M(p) \geq I(t,p)].$$

Nyní si popišme definované výrazy slovně. Proveditelnost nám říká, zda je možné vybraný přechod provést či nikoliv. Právě provádění přechodů zajišťuje určitou dynamiku sítě. Ke zjištění proveditelnosti přechodu musíme zkontrolovat, že v každém místě z množiny ${}^\bullet t$ přechodu t se nachází minimálně tolik tokenů, kolik je násobnost hrany $I(t,p)$. A když je přechod proveditelný, můžeme přistoupit k jeho provedení. Jak vidíme v definici 2.5, provedení přechodu spočívá v tom, že ze vstupních míst $I(t)$ odebereme počet tokenů zadaných násobností $I(t,p)$ a do výstupních míst $O(t)$ přidáme tokeny dle



Obrázek 1: Provedení přechodu v Petriho síti.

požadované násobností $O(t,p)$. Tímto krokem se změnil stav sítě před provedením označený M na nový stav po provedení označený M'.

Jak provedení přechodu vypadá zakreslené graficky, můžeme vidět na obrázku 1.

Zda-li můžeme provést nějaký přechod nám říká **proveditelnost sítě**.

Definice 2.6 *(převzato z [2])*
***Petriho síť je proveditelná(Enable)**, není-li při aktuálním značení M množina proveditelných přechodů $E(M)$ prázdná $E(M) \neq \emptyset$.*

Mezi další využívané vlastnosti Petriho sítí patří dosažitelnost, reversibilnost, uzamčení, živost a mrtvost.

Definice 2.7 *(převzato z [2])*
***Značení M' je dosažitelné (reachable)** ze značení M, jestliže existuje posloupnost přechodů σM, která je proveditelná ve značení M a která převádí Petriho síť ze značení M do značení M', tj.*

$$(\exists \sigma_M \in T^*)(M \rightarrow^{\sigma_M} M').$$

S pomocí dosažitelnosti můžeme pro značení M najít množinu dosažitelných značení.

Definice 2.8 *(převzato z [2])*
***Množina dosažitelnosti (reachability set) PN-systému** $\langle P, T, I, O, M_0 \rangle$ je množina $RS(M_0)$ definovaná induktivně takto:*

- $M_0 \in RS(M_0)$,
- $M \in RS(M_0) \wedge (\exists t \in T) \left\{ M \rightarrow^t M' \right\} \Rightarrow M' \in RS(M_0)$.

Definice 2.9 *(převzato z [2])*
***Značení** M Petriho sítě $\langle P, T, I, O, M_0 \rangle$ se nazývá **vždy dosažitelným (home state marking)**, jestliže je dosažitelné z každého dosažitelného značení, tj. platí-li*

$$(\forall M' \in RS(M_0)) M \in RS(M').$$

Definice 2.10 *(převzato z [2])*
PN-systém $\langle P, T, I, O, M_0 \rangle$ *se nazývá* ***reversibilní (reversible)****, je-li počáteční značení vždy dosažitelné, tj.*

$$(\forall M \in RS(M_0))M \in RS(M).$$

Definice 2.11 ***PN-systém*** $\langle P, T, I, O, M_0 \rangle$ *se nazývá systémem* ***bez uzamčení (deadlock-free)****, jestliže z počátečního značení* M_0 *není dosažitelné žádné značení, ve kterém není žádný přechod proveditelný, tj. platí-li*

$$\neg(\exists M \in RS(M_0))[E(M) = \emptyset].$$

Definice 2.12 *(převzato z [2])*
Místo p PN-systému se nazývá k-omezené (k-bounded)*, jestliže pro každé dosažitelné značení je počet tokenů v tomto místě nanejvýše rovný* k*, tj.*

$$(\forall M \in RS(M_0))[M(p) \leq k].$$

Definice 2.13 *(převzato z [2])*
PN-systém se nazývá k-omezený*, jestliže všechna jeho místa jsou k-omezená, tj. platí-li*

$$(\forall p \in P)(\forall M \in RS(M_0))[M(p) \leq k].$$

Definice 2.14 *(převzato z [2])*
PN-systémy se nazývají bezpečnými (safe)*, jestliže jsou 1-omezené dle definice 2.13.*

Definice 2.15 *(převzato z [2])*
Přechod t ***je mrtvý při značení*** M*, jestliže přechod není proveditelný v žádném značení dosažitelném ze značení* M*, tj. platí-li*

$$(\forall M' \in RS(M))[t \notin E(M)].$$

Definice 2.16 *(převzato z [2])*
Přechod t ***je živý při značení*** M*, jestliže není mrtvý v žádném značení dosažitelném ze značení* M*, tj.*

$$(\forall M' \in RS(M))(\exists M'' \in RS(M')\,[t \in E(M'')]).$$

Definice 2.17 *(převzato z [2])*
Přechod t ***je v daném PN systému mrtvý****, je-li mrtvý při počátečním značení, tj. platí-li*

$$(\forall M' \in RS(M_0))[t \notin E(M)].$$

Definice 2.18 *(převzato z [2])*
Přechod t ***je v daném PN systému živý****, je-li živý při počátečním značení, tj. platí-li*

$$(\forall M \in RS(M_0))(\exists M' \in RS(M)\,[t \in E(M')]).$$

Definice 2.19 *(převzato z [2])*
***PN systém je mrtvý**, jsou-li všechny jeho přechody mrtvé,*

$$(\forall t \in T)(\forall M' \in RS(M_0))[t \notin E(M)].$$

Definice 2.20 *(převzato z [2])*
***PN systém je živý**, jsou-li všechny jeho přechody živé, tj. platí-li*

$$(\forall t \in T)(\forall M \in RS(M_0))(\exists M' \in RS(M)\,[t \in E(M')]).$$

Definice 2.21 *(převzato z [2])*
*Obyčejnou **Petriho síť nazveme správnou (korektní) sítí**, jestliže je současně bezpečná, živá a reverzibilní, tj. platí-li současně:*

- *reversibilnost dle definice 2.10*
  $(\forall M, M' \in RS(M_0))(\exists \sigma \in T^*)\,[M \rightarrow^{\sigma} M']$,
- *bezpečnost dle definice 2.14*
  $(\forall p \in P)(\forall M \in RS(M_0))[M(p) \leq 1]$,
- *živost dle definice 2.20*
  $(\forall t \in T)(\exists M, M' \in RS(M_0))\,\left[M \rightarrow^{t} M'\right]$.

2.4 Hierarchické Petriho sítě

Doposud jsme se pohybovali pouze na jedné úrovni abstrakce. Ale představme si situaci, kdy chceme modelovat výrobní proces. Máme vstupní místo polotovarů, z něhož se hranou přesouvá polotovar do přechodu reprezentujícího výrobu a z něj je poté do místa představující sklad přesunut hotový výrobek. Chceme-li však zjistit, jaký je proces výroby, pak potřebujeme druhou nezávislou síť. Když bychom už měli velkou síť s několika desítkami takovýchto podprocesů, které každý mají rovněž nějaké podprocesy, pak už je tento způsob modelovaní nevhodný. Pro tyto účely bylo zavedeno rozšíření Petriho sítí o možnost tvořit hierarchickou strukturu pomocí substituce bloků podprocesů za elementy sítě.

Tato možnost je v programu vyřešena vkládáním samostatných procesů, kterým se poté definují vstupní a výstupní elementy a v samostatné kartě můžeme vytvořit vnitřní strukturu.

2.5 Časované sítě

Budeme-li chtít modelovat nějaký proces z reálného světa, musíme zahrnout i časové aspekty takového procesu. Existuje řada způsobů, jak časový aspekt řešit. V našem případě budeme řešit Petriho síť s časovanými tokeny.

V takovém případě má každý token vlastní časovou značku θ_t, která udává v jakém čase nejdříve je možné token provést. Dále každý přechod má nastavené nezáporné číslo, θ_t jež udává čas, který říká, jak dlouho určitý přechod trvá.

Síť má dále nastaveny své globální hodiny, které říkají, v jakém čase se simulovaný proces nachází. Nemůžeme-li žádný přechod provést, ale stále jsou v síti přechody, které jsou v budoucnu proveditelné, globální hodiny jsou posunuty kupředu na nejbližší možný čas.

Při provedení přechodu je poté výstupním tokenům nastavena časová značka na globální čas navýšený o dobu trvání přechodu.

2.6 Objektově orientované sítě

Již z bodu 3 zadání této práce je zřejmé, že předmětem této práce je simulace objektově orientované Petriho sítě (dále OOPN). To v sobě skloubí schopnost Petriho sítě přirozeným způsobem modelovat paralelní a distribuované systémy s objektově orientovaným přístupem k objektům a jejich dekompozici do tříd. Tím nám OOPN umožňují snadno modelovat rozsáhlé systémy.

Jednotlivé objekty v síti mohou představovat objekty reálného světa v modelované situaci. Každý objekt v síti má své vlastní informace a jeho hodnoty nemůže jiný objekt měnit. Ke změnám hodnot dochází pouze provedením přechodu. Způsob, jakým změna probíhá je zde nastaven skriptem.

3 Využívané technologie

3.1 Nástroj pro sazbu LaTeX

Ke psaní této práce byl využit LaTeX a oficiální šablona pro závěrečnou práci od doc. Mgr. Jiřího Dvorského, Ph.D. K editaci práce byl použit program TeXstudio a překlad prováděl MiKTeX.

Následné informace pocházení z [12, 13].
Systém LaTeXvznikl jako rozšíření systému TeX. Oba jsou to systémy určené pro profesionální sazbu dokumentů. Nejprve byl v 70. letech 20. století Donaldem Knuthem vyvinut značkovací jazyk TeX. Značkovací proto, že do obyčejného dokumentu jsou vkládány speciální značky určující způsob sazby. Tento jazyk se ukázal velice mocný, ale pro běžné uživatele byl příliš složitý. Z toho důvodu 80. letech Leslie Lamport vytvořil skupinu maker zastřešující běžně používané konstrukce TeXu a množinu základních šablon a celé to nazval LaTeX.

Jazyky TeX i LaTeXnejsou interaktivní jazyky, což znamená, že na rozdíl od psaní dokumentu například v programu Microsoft Word nevidíme během psaní výsledek v reálném čase. Abychom si mohli výsledek prohlédnout, musíme napsaný kód nejprve nechat zkompilovat. Obrovskou výhodou je, že si nemusíme hlídat styly během naší práce, protože používaná makra se vždy stylizují stejně. Proto se nám nestává, že když kopírujeme text z jiného zdroje, nekopíruje se včetně stylu, který nám často rozhodí celý dokument. Další výhodou je možnost tvorby vlastních maker, což při častém použití některé konstrukce velmi šetří čas. Nemalou výhodou je také to, že LaTeXje multiplatformní a nemáme tedy problém s přenositelností mezi různými operačními systémy.

3.2 Java

Pro implementaci je použít interpretovaný, objektově orientovaný programovací jazyk.

Následní informace pocházení z [7, 8].
Ten byl vytvořen Jamesem Goslingem ze Sun Microsystems roku 1991. Pro veřejnost byl společností uvolněn roku 1995. Společnost Sun Microsystems byla roku 2010 i s jazykem koupena společností Oracle.

Od roku 2006 byly zdrojové kódy jazyka uvolněny pod GNU licencí. Společnost Oracle pokračuje v tomto projektu, který je označen OpenJDK.

Postupem času z programovacího jazyka vznikla celá platforma, která umožňuje psát i v jiných jazycích a stále takový program spouštět pod Java virtuálním strojem.

Nejnovější verzí programovacího jazyka je verze 8, jejíž dokončení bylo plánováno na 18.3.2014 [9]. Společností Oracle byla představena o týden později [10]. Z tohoto důvodu celý vývoj probíhal ještě na staré verzi 7.

Výhodou Javy je její způsob překladu, kdy se zdrojové kódy překládají do spustitelného mezikódu nazývaného Bytecode. Tento je na cílovém počítači spuštěn ve virtuálním stroji. Virtuální stroj je součástí běhového prostředí (JRE) a dělá tak program snadno přenositelný bez nutnosti opětovné kompilace na cílovém operačním systému.

3.3 Eclipse

K práci bylo využíváno vývojové prostředí Eclipse Modeling Tools ve verzi 4.3 Kepler. Toto prostředí nám poskytuje všechny potřebné knihovny pro vývoj GMF aplikace. Tuto verzi prostředí je možné zdarma stáhnout z `http://www.eclipse.org/`.

Následné informace pocházejí z [11].

Eclipse je komunitou vyvíjený open source software cílený na vývoj v open source vývojových platformách a zpracování softwaru po dobu jeho životního cyklu. Eclipse byl původně roku 2001 vyvinut firmou IBM. V lednu roku 2004 vznikla The Eclipse Foundation, která byla vyčleněna jako nezávislá nezisková společnost, která vede komunitu vývojářů.

Filozofie Eclipse je taková, že je vytvořeno nejjednodušší vývojové prostředí, které si poté vývojář sám rozšiřuje o potřebné pluginy až v okamžiku, kdy je potřebuje.

3.4 Graphical Modeling Framework

Grafický editor Petriho sítě byl jako výsledek předešlé diplomové práce vytvořen pomoci frameworku GMF.

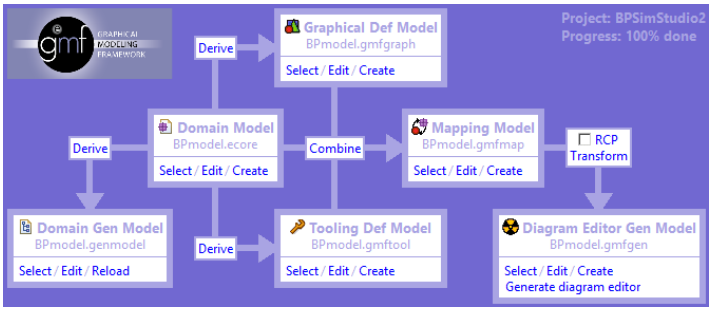
Následné informace pocházejí z [14, 15].

GMF je framework pro vytváření grafických aplikací založených na prostředí Eclipse. Tento framework v sobě spojuje dvě komponenty a těmi jsou frameworky EMF a GEF.

Eclipse Modeling Framework (EMF) je framework sloužící ke generování kódu na základě strukturovaného datového modelu. Tento framework v sobě zároveň obsahuje validátor, který nám před vygenerováním umožní kontrolu vytvořeného modelu. Po vygenerování je vytvořen model dle návrhového vzoru factory. Práce s takovým modelem poté spočívá v tom, že k tvorbě instancí jednotlivých tříd sloučí instance třídy `factory`, která je vygenerována dle návrhového vzoru singleton, ke kterému se přistupuje pomocí proměnné `eINSTANCE`.

Graphical Editing Framework (GEF) je framework sloužící k vytváření grafických editorů založených na prostředí Eclipse a sloužících pro kreslení různých diagramů.

Jak bylo zmíněno, GMF tyto dva frameworky spojuje. K tomu využívá GMF Dashboard, který si můžeme prohlédnou na obrázku 2. Práce s tímto nástrojem probíhá tak, že se tlačítkem `derive` vytvoří Domain Model. Z něj se poté vygeneruje Domain Gen Model sloužící k vygenerování EMF aplikace. Jakmile je toto hotovo, je zbylými tlačítky `derive` nutné vytvořit Graphical Def Model a Tooling Def Model. První model definuje, jak budou komponenty vypadat v diagramu, a druhý říká, jak bude vypadat paleta nástrojů. Jakmile máme toto vytvořeno, můžeme stiskem tlačítka `combine` vytvořit Mapping Model. Ten definuje, jak bude propojen EMF model s komponentami diagramu a palety, a definuje různá validační omezení. Předposlední krok je stisknutí tlačítka `Transform` a tím vytvoření poslední části, kterou je Diagram Editor Gen Model. Zde jsou nastaveny vlastnosti a chování komponent. Nakonec stiskem `Generate diagram editor` je vygenerován kód GEF diagramu nad EMF modelem.



Obrázek 2: GMF Dashboard

3.5 Groovy

Jedním z bodů zadání je použití skriptovacího jazyka pro úpravu objektů během provádění procesu. K tomu byl dle rady vedoucího zvolen sympatický skriptovací jazyk Groovy.

Následné informace pocházejí z [16, 17, 18].
Groovy je objektově orientovaný dynamický jazyk postavený na platformě Java s funkcemi podobnými skriptovacím jazykům, jako jsou Python, Ruby nebo Perl. První zmínka o Groovy se objevila již roku 2003, kdy byl navrhnut jako alternativa k jazyku Java. Plán byl vytvořit skriptovací jazyk, který poskytne sílu JVM a zároveň bude mít kratší a jednodušší syntaxi. Tento prvotní návrh byl dostatečný pro vydání JSR. Po nedlouhé době ale tento projekt jeho zastánci opustili. Vše změnil až Guillaume LaForge, který začal původní návrh záplatovat. Zároveň to byl talentovaný manager, který přesvědčil komunitu, že projekt je třeba posunout vpřed.

A tak roku 2007 vyšlo Groovy verze 1.0. Tento jazyk byl navzdory útokům ze strany původních tvůrců přijat. Přes všechny své přednosti ale Groovy trpěl několika vážnými nedostatky. Nejvážnějším nedostatkem byla rychlost, který byla i na dynamický jazyk velmi pomalá. Další vážný problém byla absence jasné filozofie a změny byly přidávány podle jejich zalíbení komunitě. Z toho důvodu bylo obtížné vývoj jakkoli předpovídat. Co však patrně nejvíce zabránilo přijetí jazyka byla jeho nedostatečné dokumentace, která se skládala pouze z jedné knihy a dvou tutoriálů.

O rok později (2008) byl vývojový tým přiveden pod společnost VMware a tím se Groovy stal jedinou alternativou Javy, za kterou stála veřejná firma. Bohužel firma se k tomuto jazyku stavěla flegmaticky a jeho vývoj tudíž pokračoval pomalu, což umožnilo vnik alternativních jazyků, které Groovy odsunuly do pozadí.

Roku 2012 vyšla verze 2.0, která vyřešila výkonnostní problémy. Zároveň v ní bylo přidáno statické typování. Díky optimalizaci generování kódu je nyní rychlost spouštění srovnatelná s nativně zkompilovaným Java kódem. Následně verze 2.1 přišla s vylepšením dřívějších výhod a plnou podporu integrace 7. verze Javy. Bohužel ale nadále přetrvává problém s nedostatečnou dokumentací.

Přes veškeré problémy Groovy překonal prvotní výkonnostní problémy a stal se jazykem připraveným k rozsáhlému použití s jedinečnými výhodami jakými jsou stručnost, čitelnost Javě podobné syntaxe, dobrou podporou unit testování, statickým i dynamickým typováním, možností meta-programování a dobrým frameworkem pro paralelní aplikace.

Na 4. čtvrtletí letošního roku 2014 je plánované[3] vydání Groovy verze 3.0, která by měla přinést podporu Javy v nejnovější 8. verzi včetně jejích nových vlastností, jako jsou lambda konstrukce, či definování defaultních metod v interface, jako to doposud bylo možné pouze v abstraktní rodičovské třídě.

3.6 Verzovací systém

Pro lepší kontrolu nad změnami a snazší návrat ze slepé uličky vývoje byl používán verzovací systém. K tomuto účelu byl zvolen systém Git. Celý průběh práce byl ukládán do lokálního repositáře, který byl následně archivován online na serveru `bitbucket.org`.

Následné informace pocházejí z [6].
Systém GIT se jako mnoho velkých věcí zrodil z vášnivého sporu. Vše začalo s vývojem operačního systému Linux. Ten byl zprvu v letech 1991-2002 vyvíjen formou záplat a archivování starých souborů. V roce 2002 začal být vývoj verzován pomocí komerčního systému s názvem Bit-Keeper. V roce 2005 se zhoršily vztahy komunity a společnosti, jež Bit-Keeper vyvíjela. Důsledkem toho bylo, že společnost přestala systém dodávat zdarma. To přimělo komunitu okolo Linuxu k vývoji vlastního nástroje dle poznatků získaných během používání Bit-Keeper. Hlavními požadavky na nový systém byly rychlost, silná podpora paralelního vývoje, jednoduchý design a schopnost efektivně spravovat i tak obrovské projekty, jako je jádro Linuxu. A tak roku 2005 vznikl Git.

Git je snadno využitelný, extrémně rychlý a efektivní systém, který umožňuje neomezené větvení pro nelineární způsob vývoje. Oproti jiným verzovacím systémům (např. Subversion nebo Perforce) je ukládání a zpracování dat odlišné. Většina systémů ukládá informace jako seznam změn jednotlivých souborů, kdežto Git ukládá snímky souborů. V zájmu zefektivnění práce jsou ukládány pouze změněné soubory a u ostatních je jen vytvořen odkaz na soubor předchozí. Další rozdíl je ten, že Gitu k většině činností stačí pouze lokální data a nemusíme tak udržovat kontakt s ostatními počítači v síti.

Další vlastností systému je to, že než je v něm cokoli uloženo, je pro změnu spočten kontrolní součet používaný k identifikaci operace. Z toho důvodu není možné libovolně měnit obsah jakéhokoli souboru aniž by si toho Git všiml. Díky tomu nemůže dojít ke ztrátě informace během přenosu, aniž by to systém nebyl schopen zjistit.

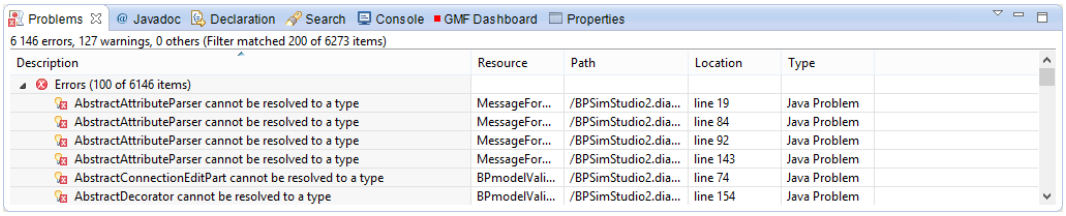
[3] zdroj http://groovy.codehaus.org/Roadmap

4 Popis výchozího stavu a analýza

4.1 První spuštění

Zjištění, že nebude třeba dělat grafický editor, ale pouze do již vytvořeného editoru udělat simulaci, bylo dobrou zprávou.

Následně po obdržení hotového editoru, který byl výsledkem již obhájené diplomové práce, nastal problém během importu do běžného prostředí Eclipse. Po importu totiž program hlásil 6146 chyb, jako na obrázku 3.



Obrázek 3: Chybějící knihovny v běžném prostředí.

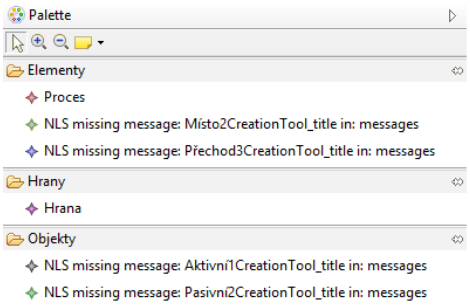
Při zkoumání chyb bylo zjištěno, že většina je způsobena chybějícími knihovnami. Veškeré pokusy o doinstalování byly neúspěšné. Důvodem toho bylo použití nejnovější verze prostředí Eclipse, jež byla stará pár dnů a knihovny k ní ještě neexistovaly. I když se počet chyb po dlouhém boji podařilo redukovat přibližně na polovinu, stále to bylo příliš. Ani po dalších pokusech se nedostavil úspěch, bylo tedy na čase hledat jiné řešení. Tím bylo navštívení stránky `eclipse.org/downloads/` za účelem stažení starší verze, stejné jako používal autor původní práce. Ale byl tady package s názvem `Eclipse Modeling Tools`, který je dle popisu určen přesně pro naše účely.

Po stažení, rozbalení a spuštění této verze prostředí Eclipse byl importován již hotový editor a počet chyb byl výrazně nižší. Při bližším zkoumání těchto chyb bylo zjištěno, že se jedná o problém s rozdílným kódováním souborů. To bylo způsobeno použitím operačního systému Linux, kde je základní písmo `UTF-8`, kdežto v projektu bylo nastaveno kódování `Cp1250`. Toto kódování editor při vytváření zdědil od operačního systému Windows, ale systém Linux ho neznal, a proto nemohl přeložit některé části kódu. Problém vyvstal hlavně díky tomu, že generátor používá k vytvoření názvů metod atribut `Title` používaný u prvků palety. Tyto názvy jsou uloženy v souboru `BPmodel.gmftool`. Jelikož tady byly atributu `Title` přiřazeny hodnoty `Místo`, `Přechod`, `Aktivní`, `Pasivní`, pak se následně při vygenerování vytvořila třída `BPmodelPaletteFactory` a v té byly vygenerovány metody, jejichž hlavičky můžeme vidět ve výpisu kódu 1.

```
private ToolEntry createMísto2CreationTool();
private ToolEntry createPřechod3CreationTool();
private ToolEntry createAktivní1CreationTool();
private ToolEntry createPasivní2CreationTool();
```

Výpis 1: Hlavičky chybně vygenerovaných metod

Po ruční opravě chybných souborů už Eclipse žádné chyby nenalezl a aplikace mohla být spuštěna. Po spuštění však bylo zjištěno, že program má problém přeložit českou diakritiku při pojmenování zmíněných prvků palety. Příčinou tohoto problému bylo opomenutí souboru `messages.properties`, který validátor neoznačil jako chybu bránící spuštění. V tomto souboru jsou uloženy výstupní texty pro editor. Proto při spuštění chyběly některé popisky v paletě nástrojů, která vypadala jako na obrázku 4.



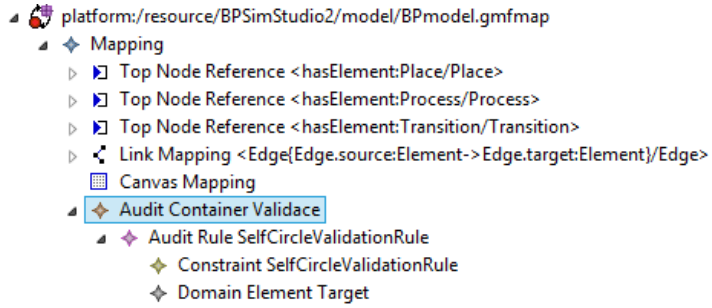
Obrázek 4: Chybně vygenerovaná paleta

Během pozdější práce bylo v kapitole 5.2 zjištěno, že tyto úpravy byly udělány zbytečně, protože při pozdějších změnách v editoru byly při vygenerování tyto změny znovu přepsány. To se projevilo až později při pokusu znovu vyhledávat česká písmenka v kódu. Proto bylo nutné přistoupil k opravě souboru `BPmodel.gmftool`. Jak již bylo míněno výše, tak právě v tomto souboru je chybně popsána paleta elementů a špatně nastavený atribut `Title`, který způsobuje problém. Proto bylo u všech problémových položek palety potřeba změnil hodnoty atributů z původních `Místo`, `Přechod`, `Aktivní`, `Pasivní` na hodnoty bez diakritiky, tedy na `Misto`, `Prechod`, `Aktivni`, `Pasivni`. Po uložení bylo nutné v okně GMF Dashboadr (na obrázku 2) kliknout na možnost `Transform` a následně použít možnost `Generate diagram editor`. Pomocí té se přegeneroval editor a projevily se změny. Nyní při pokusu vyhledávat písmenka `ř` a `í`, už Eclipse nenašel žádnou metodu s českou diakritiku ve funkčním kódu a veškeré nálezy byly pouze u textových atributů tříd a nastavení textů v `messages.properties`.

4.2 Ošetření cyklických závislostní

Další věcí, kterou bylo třeba udělat, bylo zakázat vytvoření hrany mezi objekty stejného typu (Místo->Místo, Přechod->Přechod) a zakázat vytvoření cyklické hrany, která vede do objektu, ze kterého zároveň vychází. Z principu funkce Petriho sítě jsou totiž takové hrany nedefinované. Nejprve se toto nepodařilo omezit programově, protože nebyl nalezen způsob, jak získat přístup k vytvořenému modelu. Při hledání způsobů, jak tuto problematiku řešit, bylo zjištěno, že ve frameworku GMF je možné jednoduše vytvořit `Audit`, pomoci kterého je poté požadovaný prvek validován. Vytvoření této validace je velice snadné, a proto bylo překvapivé, že již autor editoru toto neudělal.

Vytvoření validačního auditu je snadné a vytváří se v souboru `BPmodel.gmfmap`. V tomto souboru se přiřadí k poduzlu `Mapping` nový potomek `Audit Container`. V tomto kontejneru se poté vytváří potomci typu `Audit Rule`. V tom je nastavena chybová zpráva, typ chyby a živost validace (vlastnost `Use In Live Mode`). Dále kontejner uchovává 2 potomky a to `Constraint`, který říká v jakém jazyce bude validace řešena a `Element Target`, který říká, jaký objekt editoru bude validován. Vše můžeme názorně vidět na obrázku 5.



Obrázek 5: Validační audit

Jakmile je toto hotovo, je třeba kliknout na `Transform` v okně GMF Dashboadr a poté ještě upravit soubor `BPmodel.gmfgen`. V tomto souboru upravit element `Gen Diagram ProcesEditPart`, kde ve vlastnostech nastavíme možnosti `Validation Decorators` a `Validation Enabled` na hodnotu `True`.

Když je po kliknutí na `Generate diagram editor` vygenerován editor, zjistíme, že nově vygenerovaný editor není možné spustit, protože v něm byly změny, které nebyly označeny `@Generated NOT`, a proto se tyto změny přepsaly vygenerovaným kódem. Při bližším zkoumání chybových textů bylo zjištěno, že chyba vzniká pouze při pokusu vygenerovat třídu `ProcessEditPart`.

Vyřešit tento problém pomohl verzovací systém, který umožnil obnovit stav editoru před vygenerováním. U inkriminovaných metod bylo změněno označení z `@Geterated` na `@Generated NOT`, a to zajistilo, že při novém vygenerování se už tyto metody nebudou přepisovat. Po novém pokusu o vygenerování proběhlo vše úspěšně a vytvořil se funkční, spustitelný kód editoru. Zároveň přibyla třída `BPmodelValidationProvider` obsahující adaptéry pro jednotlivé validace. Adaptér obsahuje metodu `Validate`, která

byla upravena, jak můžeme vidět ve výpisu 2.

```
/**
 * @generated NOT
 */
public IStatus validate(IValidationContext ctx) {
    modelBP.Edge edge = (modelBP.Edge) ctx.getTarget();
    return (edge.getSource().getClass() == edge.getTarget().getClass()) ?
        ctx.createFailureStatus("Nedovolená cyklická závislost") :
        ctx.createSuccessStatus();
        }
```

Výpis 2: Validační metoda řešící nevalidní hrany.

Jak je z kódu patrné, celý problém byl vyřešen tak, že u každé hrany je zjišťováno, jestli je třída instance počátečního elementu rozdílná od třídy instance elementu cílového. Dále bylo potřeba nastavit, kdy se má validace spouštět.

K docílení toho byl přidán kód z výpisu 3 na konec metody `doSaveDocument` nacházející se ve třídě `BPmodelDocumentProvider`. Tím se docílilo toho, že při uložení změn v dokumentu se provedla validace, která označila chybné hrany. To problém vyřešilo jen napůl. Sice bylo jasné, která hrana je vytvořena špatně, ale stále bylo možné tuto hranu přidat.

To byl důvod, proč se znovu pustit do úprav v souboru `BPmodel.gmfmap`. Zde u auditu řešícího validaci hran byla nastavena ve vlastnostech hodnota pole `Use In Live Mode` na `True` a znovu vygenerován kód editoru. Nyní po spuštění nebylo možné chybnou hranu přidat, protože díky validaci v live modu se validace provádí během vytváření prvku a pokud tento není validní, tak je zabráněno jeho vytvoření.

```
ValidateAction.runValidation((View) document.getContent());
```

Výpis 3: Volání validace.

4.3 Další pravidla validace

Při vytváření validačních pravidel bylo potřeba vzít v potaz formální zápis Petriho sítě. V tom jsou hrany a přechody dány každé vlastní množinou a v množině se nemohou vyskytovat dva stejné záznamy. Proto bylo vytvořeno pravidlo, že každé místo a každý přechod musí mít vyplněné své unikátní jméno. To je důležité především z důvodu budoucího zpracování simulace a analýzy pomocí statistických metod.

Poslední přidané validační pravidlo bylo pro element `Process`. Zde sice nebylo potřeba nic validovat, ale pravidlo je využíváno k uložení odkazu na instanci hlavního procesu. V tomto procesu se nachází uložená, celá námi vytvořená síť se všemi hodnotami. K vytvoření tohoto validačního auditu bylo přistoupeno hlavně z důvodu, že se ani po dlouhém procházení návodů a rad na internetu nepovedlo najít způsob, jak jinak k tomuto vytvořenému objektu přistoupit kdykoliv a odkudkoliv. Za tímto účelem vznikla třída `PesInstance`, implementována podle návrhového vzoru Singleton. Tento postup byl zvolen především z časových důvodů, protože bez instance vytvořeného modelu nebylo možné začít programovat simulaci.

4.4 Úprava menu

Úprava menu započala nalezením třídy `DiagramEditorContextMenuProvider`. Tato třída vytváří kontextové menu po kliknutí pravým tlačítkem myši v oblasti vykreslování sítě. Toto vytvoření zajišťuje metoda `buildContextMenu`. Této metodě je v parametru předán objekt menu a do něj můžeme vkládat vlastní akce. K tomu v objektu menu slouží metoda `add`, která má jako přípustný parametr instanci objektu, jež má implementován interface `IAction` nebo `IContributionItem`. Zároveň bylo možné si zde všimnou již vytvořené vlastní položky pro vymazání elementu, která je řešena třídou `DeleteElementAction`.

Dalším krokem bylo vytvoření vlastní třídy `RunElementAction`, která implementuje interface `IAction`. Po nutném vygenerování neimplementovaných metod interfacu, kterých bylo neúměrné množství ve srovnání s počtem metod implementovaných ve výše zmíněné třídě řešící mazání elementů. Proto bylo lepší je vymazat a ve třídě místo interfacu nastavit dědičnost ze třídy `Action` z balíčku `org.eclipse.jface.action`. Po tomto zásahu byla třída validní a mohla být vytvořena její instance a pomoci zmíněné metody `add` tato instance přidána do menu. Po spuštění projektu nyní v kontextovém menu přibyl jeden prázdný řádek. Nyní zbývalo tuto položku nějak nastavit. Z kódu operace pro mazání se dá vyčíst, že během vytváření akcí je volána metoda `init`, která vše nastavuje. Zkopírováním této metody z akce pro mazání a přenastavení textů a ikonky dle potřeby bylo docíleno vytvoření požadovaného výsledku.

Další na řadě bylo přidání metody `run`. Ta přepisuje metodu `run` v rodičovské třídě. Že se metoda volá a provádí při kliknutí na položku v menu, bylo snadné si ověřit umístěním příkazu pro výpis textu na consoli do těla této metody. Když bylo ověřeno, že je metoda volána, mohlo dojít k připojení simulace.

4.5 Programování simulace

S přístupem k instanci hlavního procesu a k tlačítku pro vyvolání akce mohlo začít programování samotné simulace. Plán byl zvolen takový, že simulace bude programována od nejjednoduššího přesouvání objektů pouze na základě násobnosti hran a poté bude simulace rozšířena o scénáře, omezení a skripty.

Začalo tedy programování přesouvání objektů, ale hned v úvodu při jakéhokoli pokusu něco změnit tento pokus končil chybou `java.lang.IllegalStateException: Cannot modify resource set without a write transaction`. Jelikož ani po dlouhém hledání v diskuzích se nedařilo najít způsob, jak tuto chybu obejít, bylo přistoupeno k vytvoření editovatelné kopie sítě a s tou poté pracovat. To začalo tím, že byly vytvořeny balíčky `pn.model` a `pn.model.impl` přesně tak, jako byly vytvořeny v původním modelu. Do těchto balíčků bylo vytvořeno rozhraní a implementace tříd stejně, jako jsou definovány v modelu. Dále vznikla třída `PesParser`, která měla za úkol vytvořit kopii procesu získaného při validaci.

Takové kopírování spočívalo v procházení elementů hlavního procesu a vytváření instancí nově vytvořených tříd a k naplňování jejich atributů stejným způsobem. Zde ovšem nastal problém, protože v původním procesu některé prvky obsahují list dalších

prvků, kde každý z nich má atribut ukazující na instanci třídy, která ho má v listu. Tato cyklická závislost byla opomenuta, a proto program havaroval z důvodu nedostatku paměti, když se cyklicky vytvářely stále nové objekty.

Tento problém vyřešilo to, že pro každou třídu modelu byla vytvořena hashmapa, která jako klíč měla instanci původního procesu a v hodnotě udržovala lokální objekt. Poté se před vytvořením nové instance nejprve zjistí, zda taková instance již existuje, a pokud ne, pak je vytvořena nová. Po této úpravě se již kopie vytváří bez vážnějších problémů.

Když bylo možné s modelem pracovat a měnit jeho hodnoty, mohla začít práce na samotné simulaci. V modelu všechny elementy vlastní hlavní proces a uchovává je společně v jednom listu `hasElement`. Proto při vytvoření simulace dochází k iterování tohoto listu a elementy jsou rozděleny do menších listů na místa a přechody.

Nyní mohlo začít programování přesouvání objektů. V původním modelu všechny elementy vlastní hlavní proces a uchovává je společně v jednom listu `hasElement`. Proto byl tento list iterován a elementy rozděleny do listů na místa a přechody.

Aby bylo možné provádět síť náhodně, bylo nutné vytvořit metodu, která z listu přechodů vrátí jeden náhodně vybraný přechod. Tento přechod je předáván jako parametr metodě `provedPrechod`. Tady se vyskytl problém při pokusu projít všechny hrany tohoto přechodu, které by dle analýzy měly být uloženy v poli `linked`, ale to bylo prázdné. Kvůli obavám udělat jakýkoli zásah do modelu, bylo nutné řešit problém jinak. Řešením se ukázalo být doplnění jedné podmínky v místě vytváření listů pro místa a přechody. Pomocí podmínky navíc se zde začal tvořit i list hran.

Když bylo určeno, který přechod se bude provádět a zjištěno, jaké jsou v mapě hrany, mohla začít implementace pohybu objektů. Nejdřív byla řešena práce na vstupních hranách. Takovou hranu během procházení listu hran poznáme podle toho, že prováděný přechod má nastavený v atributu `target`. Dále bylo u hrany z atributu `source` načteno místo, z kterého se budou objekty brát. Určující atribut udávající typ objektu je atribut name, který je odkazem na instanci třídy `GroupName`. Hrana má definovanou svou násobnost pomocí listu instancí třídy `Group`, která udává, kolik objektů majících přiřazený typ daný instancí `GroupName` se bude z místa brát. Samotné odebírání objektu probíhalo odečtením počtu odebraných od počtu objektů v místě.

Takto naprogramovaná simulace by však umožňovala počet objektů zmenšovat až do záporných hodnot, což je špatně. Proto byla vytvořena metoda `isProveditelny`, která zjišťuje proveditelnost přechodu. Porovnává počet objektů v místě s počtem přesouvaných, a pokud bychom chtěli přesouvat více objektů, než se v místě nachází, pak návratovou hodnotou `false` prohlásí přechod za neproveditelný. Jako další vznikla metoda `getProveditelne`, která iteruje list přechodů a vrátí list pouze těch, které jsou proveditelné. Poté je náhodný přechod simulace vybírán pouze z listu těch proveditelných.

Když byla vytvořena takto fungující simulace vstupních hran, započalo programování hran výstupních. To spočívalo v tom, že v místě, kde probíhá testování, jestli je hrana vstupní, byl přidán blok `else`, který řešil hranu výstupní.

Do tohoto bloku byl zkopírován kód z bloku pro hranu vstupní a upraven tak, aby se místo bralo z atributu `target` a namísto odečítání se objekty do místa přičítaly. Problém

však vyvstal v okamžiku, kdy v cílovém místě objekt ještě neexistoval. V takovém případě bylo nutné novou instanci objektu vytvořit a vložit ji do cílového místa. Zde nastal daleko větší problém pramenící ze špatné analýzy a návrhu současného modelu. Za současného stavu nebylo možné zjistit, zda nový objekt je aktivní, nebo pasivní. Jediné, co při tvorbě objektu víme, je jen to, jaká instance třídy `GroupName` je mu přiřazena. Jediná možnost, jak toto pouze z názvu poznat, by byla naprogramování pevného slovníku, nebo umělé inteligence rozpoznávající význam z názvu typu třídy. Z časových důvodů byl prozatím tento problém vynechán a jako nový byl vždy vytvářen aktivní objekt.

4.6 Vytvoření requestu

S hotovou nejjednodušší formou simulace bylo nutné začít hledat způsob, jak se vyhnout chybě při pokusu o změnu vykreslené mapy. Na oficiální stránce helpu platformy Eclipse [22] se píše, že vykreslený model je přístupný pouze pro čtení. Chceme-li model měnit, je potřeba vytvořit požadavek (`Request`). Po několika neúspěšných pokusech o vytvoření requestu dle různých tutoriálů bylo usouzeno, že zvolený postup je špatný. Nový postup byl pokusit se z kódu zjistit, jak funguje již vytvořený příkaz pro vymazání. Tady byl předpoklad, že požadavek na vymazání musí v sobě obsahovat obecný požadavek.

Začalo tedy procházení kódu a zjišťování, jak třídy požadavku `RunElementAction` postupně vznikají rozšiřováním svých rodičů. Nejprve dědí ze třídy `DefaultDeleteElementAction`. Ta je programovou částí svého abstraktního rodiče `AbstractDeleteFromAction`. Tato abstraktní třída už je potomkem obecné třídy `DiagramAction`, což je ona obecná třída, kterou potřebujeme.

Proto byla změněna rodičovská třída dříve vytvořené třídy `RunElementAction` z `Action` na `DiagramAction`. Po této akci bylo třeba dodefinovat chybějící metody. Těmi jsou `isSelectionListener` a `createTargetRequest`. Obsah těchto metod byl zprvu zkopírován ze třídy `AbstractDeleteFromAction`, která je jednou z rodičovských tříd původního příkazu pro mazání. Když byl v tomto okamžiku program spuštěn, pak při použití akce `RunElementAction` tato mazala objekty sítě, jako původní metoda pro mazání. Akce, jakou request vykonává, je definována jednou z přidaných metod. Jmenovitě metodou `createTargetRequest`, na kterou se můžeme podívat ve výpisu 4.

```
protected Request createTargetRequest() {
    return   new EditCommandRequestWrapper(new DestroyElementRequest(getEditingDomain
        (), false));
}
```

Výpis 4: Vytvoření requestu pro mazání.

Jak vidíme, metoda vytváří pouze novou instanci třídy `DestroyElementRequest`, která je umístěná v balíčku `org.eclipse.gmf.runtime.emf.type.core.requests`. Stačilo tedy začít procházet tento balíček a zjistit, jaké další třídy jsou v tomto balíčku uloženy. Byla zde nalezena třída `SetRequest`. Dle dokumentace, kterou pro tuto třídu zobrazil našeptávač, se právě tato třída používá ke změně. Do konstruktoru musíme třídě předat tři argumenty a těmi jsou instance objektu, který chceme měnit. Druhý argument říká, jakou vlastnost měníme a třetím zadáváme novou hodnotu. Když se po proběhnutí

této metody zavolá metoda `run` v rodičovské třídě, provede se požadovaná změna. Aby bylo možné obecně vytvářet požadovaný request v metodě `run`, byly pro tyto tři vstupy vytvořeny lokální proměnné, které bylo poté možné v metodě `run` nastavit, a při vytvoření requestu zavoláním metody `createTargetRequest` jsou tyto hodnoty převzaty z nastavených proměnných.

Jakmile bylo vytvoření requestu připravené, mohl být načten singleton instance třídy `PesInstance`, a nad tím v metodě `run` zkusmo procházeny vytvořené elementy a měněny jejich názvy. Po prvních nezdarech, kdy se měnil pouze jeden prvek, bylo zjištěno, jak request funguje. Že nestačí pouze volat pro každou změnu metodu `run` v rodičovské třídě, ale také je třeba před každým zavoláním této metody vytvořit nový request s aktuálními daty, a ten poté nastavit rodičovské třídě. K tomu slouží metoda `setRequest` v rodičovské třídě, do které je předáván výsledek lokální metody `createTargetRequest`.

Nepříjemností byla výjimka `NullPointerException`, která se vypisovala do console během načítání a při každém pokusu něco měnit před provedením validace. Nicméně kromě vypisovaní stack trace do console nic jiného tato chyba nezpůsobovala, a proto byla prozatím ignorována. V okamžiku, kdy už byla provedena validace a instance třídy `PesInstance` přestala mít hodnotu `null`, chyba zmizela a po kliknutí na příkaz se měnila jména, přesně jak bylo naprogramováno. Vyřešení tohoto problému je později věnována kapitola 4.8.

4.7 Vykreslování výsledků

Jakmile byl nasimulovaný základní pohyb a byl vytvořen request, kterým je možné změny vykreslit, bylo nutné tyto dvě věci spojit a tím promítnout v simulaci provedené změny na obrazovku. To nebylo zase tak složité a na kód se můžeme podívat ve výpisu 5.

```
Process proc = PesInstance.getInstance().getMainProc();
if (proc != null) {
    PesParser pars = PesInstance.getInstance().getParser();
    for (Element e : proc.getHasElement()) {
        if (!( e instanceof Place))
            continue;
        Place p = (Place) e;
        if (p.getHasObject() != null)
            for (Object o : p.getHasObject()){
                target = o;
                targetValue = ModelBPPackage.eINSTANCE.getObject_Count();
                newValue = ((Object)pars.getLokalObject(o)).getCount;
                super.setTargetRequest(createTargetRequest());
                super.run(progressMonitor);
            }
    }
}
```

Výpis 5: Primitivní simulace elementů.

Nejprve bylo potřeba ověřit, že je přístup k instanci hlavního procesu. Tato instance je načítána ze singletonu třídy `PesInstance` a poté je ověřováno, jestli už má nastavený hlavní proces, nebo ještě stále přístup k modelu nemá. Pokud je zjištěno, že přístup k mo-

delu je již k dispozici, je vytvořena lokální proměnná parseru, která v sobě má uložené HashMapy uchovávající pro každý objekt vykresleného modelu jeho kopii, nad kterou byla prováděna simulace. Následně jsou iterativně procházeny elementy modelu a hledají se místa. Když je element místa nalezen, jsou u něj procházeny jeho objekty a pro každý z nich vytvořen nový request, kterému je jako objekt ke změně předáván aktuální objekt `o`. Jako cílová proměnná, která bude měněna, je načtená ze singleton instance třídy `ModelBPPackage` hodnota `getObject_Count`, která requestu říká, že bude měněná hodnota `count` v objektu třídy `Object`. Poslední věcí, kterou je potřeba předat, je nová hodnota měněné proměnné `count` a ta je získána z kopie objektu `o`, kterou lze získat parserem pomocí metody `getLokalObject`. Zbývá už jen v rodičovské třídě zavolat `run` a požadovaná změna je provedena. Po spuštění takto naprogramované simulace již program po zavolání akce úspěšně mění počty objektů. Jen zatím neumí přidat nové, když ještě v místě žádný objekt není.

Bylo třeba tento nedostatek odstranit. Dobrým řešením se zdálo být přestat měnit pouze atribut `count` pro každý objekt v místě, ale rovnou pro místo přenastavit celé pole. Toho je možné docílit tím, že jako `target` nebude předáván objekt `o`, ale místo `p`. Poté je do proměnné `targetValue` nastavena hodnota `ModelBPPackage.eINSTANCE.getObject_Count()` a jako nová hodnota je vloženo pole vniklé při simulaci. Bylo ale opomenuto, že kopie jsou instance zkopírovaných a upravených tříd, ale debuger to po spuštění pokusu o akci připomenul.

Po uvědomění si, že vytváření kopií vlastních tříd bylo kontraproduktivní, byly tyto třídy vymazány. Jelikož byly kopie tříd z modelu, pak po vymazání stačilo pouze upravit importy a aplikace zase fungovala. Tyto zbytečné třídy vznikly hned na začátku práce v důsledku nedostatečné znalosti tvorby GMF programu a vytváření pluginu pro Eclipse. Celá tato situace vznikla z obavy, aby se práce nezasekla nad něčím, co třeba nebude nakonec vůbec potřeba a na důležité věci poté nezbude čas. Hlavním důvodem byla neznalost, že import referencí u pluginu probíhá jinak, než u běžné aplikace. Takže i přes přidání projektu pro model do Build path se nedařilo dostat ke třídám definovaných v modelu této aplikace. Tak byla utvořena jejich kopie a alespoň základní kostra simulace. Až během práce na validaci byl nedostatek napraven a zjištěno, že reference u Eclipse pluginu se nenastavují v Build path, ale v souboru `MANIFEST.MF`. Po úpravě vlastnosti Required Plug-ins přidáním projektu `BPSimStudio2` byl získán přístup k třídám hlavního modelu.

Po těchto úpravách už vykreslování simulace fungovalo. I nové objekty se vytvářely, ale stále to nebylo dobře, protože objekty neměly vyplněné všechny proměnné. Hlavně byla chybně vyplněná proměnná `contained`, která říká, pod kterým elementem bude objekt uložen. Ta byla nastavená na kopii místa. To způsobovalo, že stav po simulaci nebylo možné uložit, protože končil chybovým oknem. Po úpravě nastavování této proměnné již byla simulace v pořádku a vykreslovala se jak měla.

4.8 Oprava chyb s requestem

S přibývajícím kódem bylo stále nejpříjemnější, že request stále vypisuje chyby, dokud nemá nastavený model a hlavně, že toto rozdvojené pracování s modelem a jeho kopií

bylo hodně nepřehledné a práce s ním byla dost náchylná na chyby a zpětně nepřehledná. Zároveň se při každém novém vykreslení zbytečně všechny objekty tvořily nové a staré se zahazovaly. Bylo třeba vyřešit, jak to udělat, aby se změny prováděné v simulaci dělaly nad vykresleným modelem a nikoli nad jeho kopií. Toho bylo možné docílit jen tak, že pro každou změnu bude hned vytvořen request, který ji bude měnit rovnou i na obrazovce.

Jako řešení se nabízelo přesunout celý request do projektu k simulaci a v menu volat pouze obecný `IAction`, který teprve bude volat simulaci, která by si requesty řídila sama. Proto byla třída `RunElementAction` zkopírována do svého projektu simulace a přejmenována na `ChangeAction`. Poté byla třídě `RunElementAction` změněna rodičovská třída z konkrétní `DiagramAction` na obecnou `Action`. I když nový rodič nepožadoval instanci `IWorkbenchPart` jako součást konstruktoru, bylo třeba tuto instanci předat dál do nové třídy v simulaci, protože tato instance requestu říká, kde se nacházejí objekty, které má měnit.

Ze simulace byl vymazán již zbytečný parser a začala oprava simulace. K tomu, aby zase fungovala bylo nutné vytvořit několik dalších metod. Předně bylo třeba udělat metodu `updateObject`. Tato metoda pouze volá dříve upravenou metodu `run` třídy `ChangeAction`. Tuto třídu je možné vidět v příloze ve výpisu 25. Mnohem zajímavější je druhá přidaná metoda `updatePlaceAddObject`, která je ve výpisu 6. Tato metoda obstarává přidávání nového objektu do místa.

```
ChangeAction action;
  ...
private void updatePlaceAddObject(Place plac, Object obj) {
    EList<Object> list  = plac.getContains();
    List<Object> listNew = new ArrayList<Object>();
    for  (Object object  :  list ) {
        listNew.add(object);
    }
    listNew.add(obj);
    action.run(plac,  ModelBPPackage.eINSTANCE.getPlace_Contains(), listNew);
    action.run(plac,  ModelBPPackage.eINSTANCE.getPlace_HasObject(), listNew);
    action.run(obj,  ModelBPPackage.eINSTANCE.getObject_Contained(), plac);
}
```

Výpis 6: Metoda přidávající objekt.

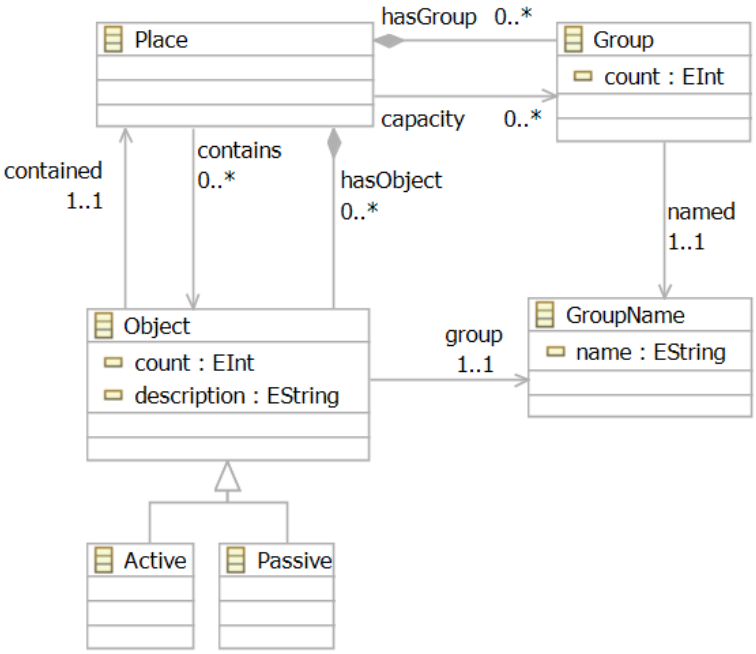
5 Přepracování modelu

5.1 Problém s modelem

Když bylo dosaženo fáze, kdy začít programovat funkcionalitu scriptu, vyskytl se problém, o kterém se v závěru práce zmiňuje i autor editoru.

"*Ještě jedna nesplněná část zadání práce zbývá, a tou je možnost přidávání atributů objektům diagramu. Důvodem nesplnění této části zadání však nebyla časová tíseň, ale opomnění při analýze. Už tento fakt napovídá tomu, že přidat tuto funkcionalitu bude znamenat zásah do celého modelu, nejspíš i nové generování kódu*"[1].

Při hlubším zkoumání tohoto problému ovšem vyvstal mnohem závažnější problém. Tím problémem bylo, že model, jak byl analyzován a navrhnut, je vzhledem k úkolu této práce naprosto nevyhovující. Problém nastal především s reprezentací míst Petriho sítě. Jak je místo navrženo, se můžeme podívat na obrázku 6.



Obrázek 6: Původní návrh místa

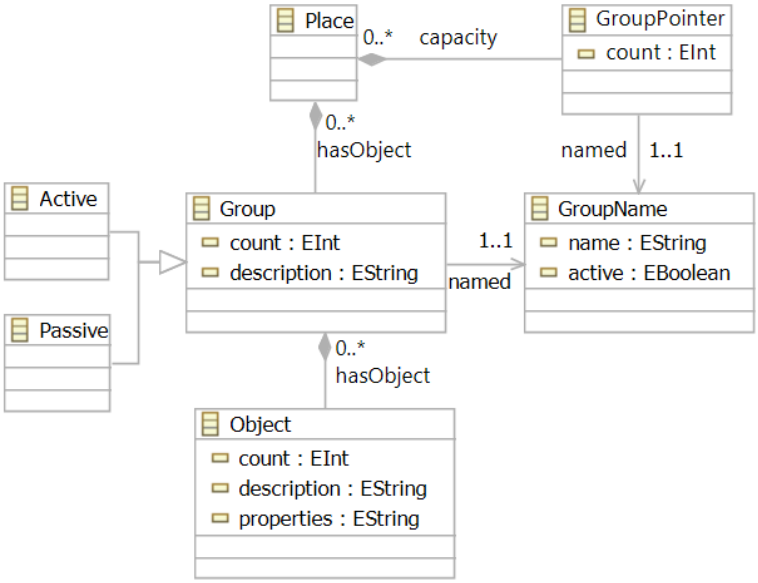
Dle zadání by měla simulace umožňovat práci s objekty různých typů a těm následně definovat uživatelské atributy. Problém však byl ten, že v editoru žádné objekty jako takové nejsou. Je zde pouze třída `Object`, která slouží jako rodičovská dvěma vykreslovaným třídám `Active` a `Passive`. Tyto vykreslované třídy rodičovské třídě nepřidávají žádnou funkčnost, ani žádné atributy. Jejich význam je pouze ten, že ve vykresleném diagramu mají jinou barvu.

Dle zadání by to ale mělo být tak, že každé místo má uložené skupiny objektů, obsahující samotné objekty a ty se poté budou přesouvat. Těmto přesouvaným objektům

bude možné pomocí skriptu v přechodu nastavovat uživatelské atributy. Bohužel současně naprogramovaný editor má v místě nastavenou skupinu objektů matoucně nazvanou `Object`. Tato skupina pouze říká, kolik má v sobě objektů a jakého jsou typu, ale samotné objekty nikde neexistují. Z místa vede hrana (`Edge`), která má pomocí listu instancí třídy `Group` nastavenou násobnost (`multiplicity`). V instanci třídy `Group` má však zase uložený pouze typ přesouvaných objektů a počet těchto objektů. Proto nebyl problém zjistit kolik a jakých objektů do přechodu a z přechodu jde. Mohla být pro tyto vstupní objekty vytvořena třída a v přechodu vytvářeny její instance dle počtu vstupních a výstupních objektů. Následně nad těmito instancemi by byly prováděny akce definované scriptem a přidávala jim uživatelské atributy, jako je požadováno v zadání. Jako minimální splnění úkolu by toto stačilo, ale nebylo by logické vytvořit objekt, který je ihned po nastavení zahozen. Proto bylo rozhodnuto splnit úkol i za cenu nutnosti zásahu do základů hotového editoru a jeho přepracování.

5.2 Změna modelu

A tak začala editace souboru `BPmodel.ecore`, kde byl model přepracován tak, aby se v diagramu skutečně pracovalo a objekty a nikoliv jen s číslem udávající počet objektů, které by vznikly při začátku provádění přechodu a po jeho skončení zanikly. Nově byl model analyzován a navrhnut, jako na obrázku 7.



Obrázek 7: Nový návrh místa

K budoucí práci je potřeba, aby v místě byly uložené skupiny dle svého typu a v nich samotné instance objektů. Nejprve byl u místa změněn typ reference `hasObject` z `Object` na `Group`. Poté u tříd `Active` a `Passive` došlo ke změně rodičovské třídy.

Následovalo vytvoření nové proměnné typu EReference ve třídě Group. Datový typ této reference byl nastaven na třídu Object. Pomocí vlastností Lower Bound a Upper Bound byla nastavena násobnost na hodnoty 0 spodní mez a -1 horní mez. Právě číslo -1 u horní meze říká, že horní mez není omezena. Další změnou bylo nastavení vlastnosti Containment na hodnotu true. Tím je docíleno toho, že třída Group je vlastníkem instancí třídy Object. U toho bylo třeba změnit datový typ atributu contained z Place na Group. Když to bylo vše hotové, mohla být u tohoto atributu nastavena hodnotu vlastnosti EOpposite na hasObject : Object což znamená, že každý objekt musí mít přiřazenou příslušnost právě k jedné skupině Group.

Nyní bylo nutné nějak nahradit původní funkcionalitu třídy Group. Pro tento účel vnikla třída GroupPointer, která plnila funkci původní třídy. Prakticky je to ukazatel na třídu určující typ prvku (GroupName), rozšířený o proměnou udávající počet prvků. Tato třída je poté využívána pro nastavení kapacity místa a určení násobnosti hrany. Tedy všude tam, kde nás zajímá s kolika a jakými objekty se bude pracovat, ale samotné instance objektů nepotřebujeme uchovávat. Když byla tato třída vytvořena, byl ji jako vlastnost EType nastaven atribut capacity třídy Place a u atributu multiplicity třídy Edge.

Po přenastavení modelu bylo nutné přejít do okna GMF Dashboard (na obrázku 2) a u položky Domail Gen Model kliknul na tlačítko Reload. Po odklikání průvodce se otevře soubor BPmodel.genmodel. V tom je z kontextového menu nad kořenem stromu zvolena možnost Generate All. To nově vygeneruje třídy modelu, balíček odkazů na atributy pro potřeby requestu, továrnu pro tvorbu instancí a rovněž kód editoru a testovací třídy. Tato změna modelu ale rozbila kód simulace. Jelikož zatím došlo jen k přetypování, oprava těchto chyb byla rychlá.

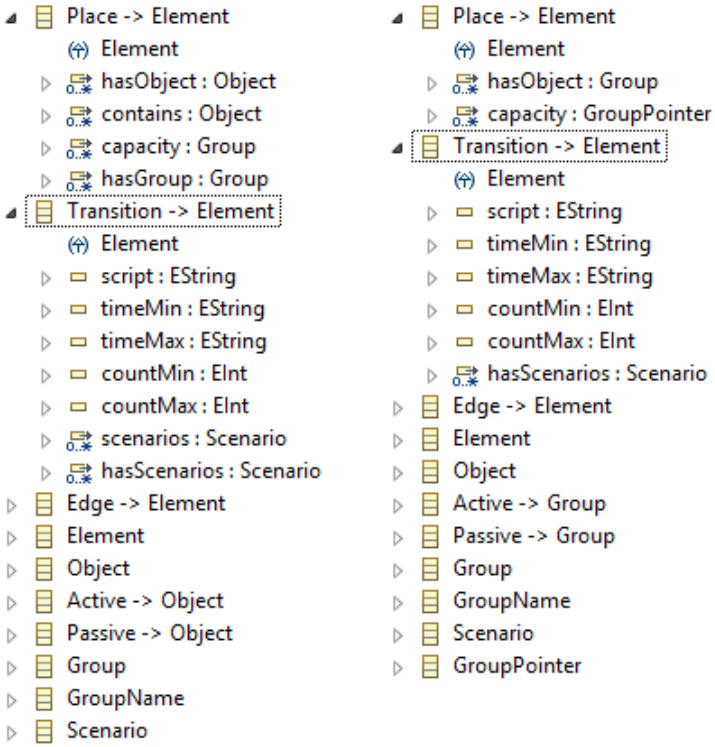
Přepracování pokračovalo úpravou dalších částí v GMF Dashboardu. Části Tooling Def Model a Graphical Def Model, které jsou definované soubory BPmodel.gmftool a BPmodel.gmfgraph, žádnou změnu nepotřebovaly. První říká jen, jak bude vypadat paleta komponent diagramu a druhý určuje, jak budou vykreslené elementy v diagramu vypadat. Problém nastal s částí Mapping Model, která je definována souborem BPmodel.gmfmap, jež při validaci hlásil chyby. Tento soubor určuje, která část modelu je mapována na kterou komponentu diagramu. Zde bylo nutné u komponent Active a Passive změnit atribut tříd, ze kterých se získává vypsaný text. Po kliknutí na Transform se změny promítnou do poslední části Dashboardu, kterou je Diagram Editor Gen Model, jež je uložen v souboru BPmodel.gmfgen. Ten při validaci žádnou chybu nenašel, a tak mohla práce pokračovat. Už stačilo jen kliknout na Generate diagram editor a tím se přepíše i projekt definující diagram.

Jakmile byl diagram nově vygenerován, bylo třeba opravit vzniklé chyby. Ty vznikly všude tam, kde bylo zasaženo do kódu a ve třídách řešících dialogová okna. Bylo proto třeba nastudovat a pochopit jejich kód a ten předělat tak, aby fungoval, jak má.

Po zjištění, jak snadné je provést změnu modelu, mohlo dojít k dalším úpravám tak, aby se s ním v simulaci dobře pracovalo. Jako první došlo k opravě atributu linked u třídy Element, která slouží jako rodičovská pro třídy Place, Transition, Process a Edge. Jak už bylo dříve zmíněno v kapitole 4.5, tento atribut nefungoval, jak by měl.

Bylo třeba nastavit vlastnost `EOpposite`, aby se automaticky plnil a udržoval stav konzistentní s vykresleným modelem. Tady se vyskytl problém, že atribut je možné párovat s dvěma možnými atributy třídy `Edge`, kterými jsou `source` a `target`. Ke správné funkci by ale bylo potřeba, aby `linked` byl spárován s oběma současně. Jelikož to nebylo možné, došlo k nahrazení atributu `linked` dvěma novými atributy `linkedIn` a `linkedOut`. Jak již názvy napovídají, nově budou vstupní a výstupní hrany ukládány zvlášť.

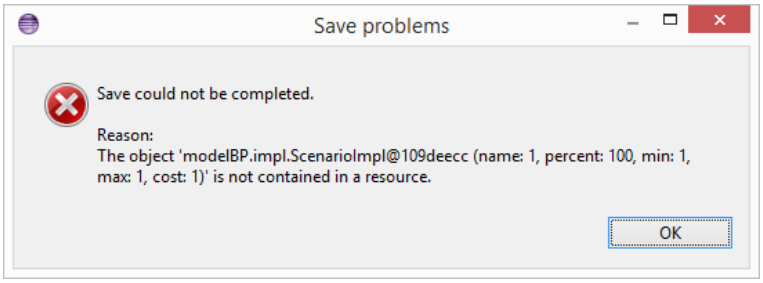
Následně došlo k odstranění z pohledu simulace zbytečně redundantních atributů, které měly třídy `Place` a `Transition`. To si můžeme prohlédnout na obrázku 8, kde je zřetelně viditelná změna.



Obrázek 8: Původní (vlevo) a opravený(vpravo) model sítě

Při vytvoření modelu po těchto úpravách bylo zjištěno, že duplicity byly patrně pouze kvůli dialogovým oknům, jejichž třídy hlásily chyby. Po změně kódu nahrazením všech odkazů na vymazané pole odkazy na ty, co zůstaly. Po spuštění vypadalo vše v pořádku a funkční, až do chvíle, než bylo otevřeno dialogové okno přechodu. Po jeho uzavření se změnil diagram, i když k žádné uživatelské změně nedošlo. Přesto prostředí ukazovalo, že došlo ke změně a je třeba diagram uložit. Při pokusu o uložení ale vyskočila chybová hláška upozorňující na to, že objekt musí být obsažen ve zdroji (obr 9).

Důvod vzniku této chyby spočívá v tom, že při každém otevření dialogového okna přechodu se pro seznam vypsaných scénářů vytváří nové instance, ale odkazy na přiřa-



Obrázek 9: Chybová hláška

zené scénáře hrany zůstávají na instanci původní a ty potom nejsou nijak ukládány. Tento problém byl vyřešen úpravou kódu třídy `TransitionAttributes`. V té je umístěna metoda `setTransitionImpl`. Tato metoda nastaví argumenty načtené z dialogového okna instanci přechodu, jíž obdrží jako argument. Bylo proto využito toho, že pořadí scénářů v listu se nemění a dialogová okna neumožňují scénáře ze seznamu vymazat. Díky tomu je možné v metodě získat původní list scénářů z předaného přechodu a tento list poté procházet a nastavoval mu atributy na hodnoty získané skrze dialogové okno. Tím bylo vyřešeno vytváření nových objektů a vyhazování chyby. Nakonec ještě probíhá jeden cyklus, který začíná na počtu prvků pole v instanci přechodu a končí na počtu prvků pole získaného od uživatele skrze dialogové okno. Během průchodu cyklu se do lokálního listu přidávají prvky z listu získaného od uživatele. Tím je zajištěno přidávání nových scénářů, které by se jinak bez této konstrukce po uzavření dialogového okna zahazovaly.

5.3 Přidání uživatelských atributů objektu

Další nutnou změnou byla potřeba doplnit chybějící možnost definování uživatelských atributů samotným objektům. Jako příhodné řešení tohoto problému bylo přidání atributu typu `EHashMap` k objektům třídy `Object`. Tato mapa by poté uchovávala uživatelem definované proměnné a jejich hodnoty. Zde vyvstal problém, že EMF neumí automaticky serializovat hashmapu. To by v důsledku znamenalo, že objekty by měly své atributy po dobu běhu programu, ale v okamžiku ukončení by se tyto hodnoty nikam neuložily a uživatel by o ně přišel.

Bylo proto nutné změnit typ těchto objektů na `EString` a zajistit převoditelnost z mapy na řetězec a zpět. Převedení objektu z mapy na řetězec nebylo složité, protože to je již implementované pomocí metody `toString`. Problém však nastal při zpětném převádění. Za tímto účelem byla vytvořena metoda, která by převod umožňovala. Tuto převoditelnost umožnila instance třídy `Properties`. Ta umožňuje vytvořit množinu objektů typu `Entry`, který představuje dvojici dvou objektů, z nichž první je klíč a druhý hodnota. Aby třída `Properties` tuto množinu zvládla vytvořit, musí mít data ve tvaru klíč=hodnota a jednotlivé záznamy oddělené novým řádkem [23]. Takto vytvořenou množinu objektů `Entry` už je možné iterovat a vytvořit z objektů mapu. Kód této

metody můžeme vidět na výpisu 7.

```
Properties props = new Properties();
props.load(new StringReader(text.substring(1, text.length() - 1).replace(",␣", "\n")));
Map<String, String> map = new HashMap<String, String>();
for (Map.Entry<Object, Object> e : props.entrySet()) {
    map.put((String)e.getKey(), (String)e.getValue());
}
return map;
```

Výpis 7: Metoda převádějící String na HashMapu

V takto napsané metodě však dochází k tomu, že jakýkoli vložený objekt po provedení vrátí jako `String`. Bylo proto třeba tento problém ošetřit a alespoň základní hodnotové datové typy zachovat. Z toho důvodu bylo třeba modifikovat kód jak pro převedení do mapy, tak i pro převod z mapy na řetězec. Pro tento účel bylo potřeba jednotlivé datové typy vzájemně odlišit. Pro tento účel bylo zvoleno vložit jako první znak řetězce hodnoty zástupný znak unikátní pro každý definovaný datový typ. Jaký znak je přiřazen jakému datovému typu můžeme vidět v následující tabulce.

Datový typ	Zástupný znak
int	I
float	F
double	D
BigDecimal	B
String	S
Object	O

Aby bylo možné takto převádět mapu na řetězec, bylo nejprve potřeba iterovat přes všechny `Entry` v mapě a zjišťovat jejich typ. Ke zjištění příslušnosti k třídě posloužil operátor `instanceOf`. Příklad jeho použití pro rozpoznání typu `double` si můžeme prohlédnout na výpisu 8. Ve stejném duchu byla opravena i metoda `toMap`, která je k vidění v příloze ve výpisu 26.

```
HashMap<String, Object> local = new HashMap<String, Object>();
for (Entry<String, Object> set : map.entrySet()) {
    ...
    if (set.getValue() instanceof Double) {
        local.put(set.getKey(), ("D" + set.getValue()));
        continue;
    }
    ...
}
return local.toString();
```

Výpis 8: Příklad zjištění typu objektu

5.4 Vytváření objektů

Jako další bylo potřeba programově vyřešit vytváření objektů v místě. Model uchovává počet objektů v proměnné `count`. Tato proměnná je snadno nastavitelná pomocí již hotového dialogového okna. Proto bylo příhodné navázat vytváření objektů na nastavování této proměnné. Za tímto účelem bylo potřeba změnit metody ve třídě `GroupImpl`. Nejprve došlo k úpravě `setCount`. Do těla té byl přidán cyklus, který vytvoří list s nastavovaným počtem instancí třídy `Objekt`. Následně je na původní list zavolána metoda `clean`, jež z něj vymaže všechny objekty a následně je pomocí metody `addAll` přidán cyklem vytvořený list. Nakonec je třeba tuto metodu označit `@generated NOT`, aby se při změně modelu nepřepsala. Po spuštění takto upraveného programu už byly objekty vytvořeny, protože metoda `setCount` je volána také při načítání uložené sítě. Při dalším spuštění se ale projevil problém, protože program první načetl hodnotu `count` a vytvořil tolik objektů, ale následně k nim ještě přidal uložené objekty. To způsobovalo, že při každém spuštění se počet objektů zdvojnásoboval. K vyřešení toho bylo třeba udělat další změnu modelu. Tou změnou bylo přenastavení vlastnosti `Transient` u proměnné `count` na hodnotu `true`. Toto nastavení zajistilo, že se proměnná `count` přestala ukládat do souboru, ale že po uzavření programu zaniká. Z toho důvodu bylo ještě potřeba pozměnit metodu `getCount` tak, aby nevracela hodnotu proměnné `count`, ale konstrukci `return getHasObject().size();` vracela velikost listu s objekty.

Další věcí, která potřebovala upravit, byla samotná tvorba objektů po změně počtu. Tady bylo nepříjemné, že při každé změně se všechny objekty vytvořily nově. To by mohl být problém, když by uživatel během simulace měnil počet objektů v místě, přišel by o uživatelské atributy nastavené stávajícím objektům. Způsob, jak tomuto problému předejít, je řešit jen rozdíl počtu prvků, jak to můžeme vidět ve výpisu 9.

```
for (int i = getHasObject().size(); i < newCount; i++) {
    modelBP.Object obj = ModelBPFactory.eINSTANCE.createObject();
    obj.setContained(this);
    obj.setDescription("generovane");
    getHasObject().add(obj);
}

if (newCount < getHasObject().size()) {
    List<modelBP.Object> nove = new ArrayList<modelBP.Object>();
    for (int i = 0; i < newCount; i++) {
        nove.add(getHasObject().get(i));
    }
    getHasObject().clear();
    getHasObject().addAll(nove);
}
```

Výpis 9: Způsob změny počtu objektů v listu

První cyklus zajistí, že pokud je hodnota `newCount` pro požadovaný počet větší než kolik je objektu, pak jsou vytvořeny nové objekty a přidány na konec listu. A druhý blok řeší stav, kdy chceme počet objektů zmenšit. Tady bylo několik možných přístupů, jak změnu provést. Tři hlavní možnosti jsou chápat list jako frontu, zásobník, nebo odebírat

objekty náhodně. V implementaci byl zvolen přístup zásobníku. To znamená, že budeme odebírat nejnovější objekty z konce listu. Jinými slovy vrátíme pouze požadovaný počet prvních objektů a co je navíc, je zahozeno. Tento přístup byl zvolen proto, aby nebyly zahazovány nejstarší objekty, u kterých je pravděpodobnější, že už mají nastavené nějaké uživatelské atributy.

6 Dokončení simulace

6.1 Úprava simulace pro nový model

Nyní, když byl model opraven, aby obsahoval objekty, bylo potřeba změnit kód simulace, aby začal pracovat s objekty. Doposud totiž program v simulaci pouze měnil proměnnou `count`. Díky vytváření a mazání objektů právě pomocí této proměnné se mohlo na první pohled zdát, že se objekty přesouvají už nyní. Ve skutečnosti ale žádný objekt nijak přesouván není, nýbrž původní objekty ve výstupních zaniknou a v cílových se vytvoří zcela nové. Aby se tomu předešlo, musel být v simulaci upraven kód metody `proved`. První krok spočíval v přesunutí objektů z místa. To probíhá tak, že je do hashMapy načten požadovaný počet objektů. Zbylé objekty jsou uloženy do nového listu a tento list je potom předán jako vstup do metody `updateGroup`, která vytváří requesty. Ukázku kódu můžeme vidět na výpisu 10.

```
HashMap<String, Object> pracovniEle = new HashMap<String, Object>();
...
int mapI = 0;
for (GroupPointer obj : edge.getMultiplicity()) {
    Group target = objMista.get(obj.getNamed()); //ziska z mista pozadovanou skupinu
    ArrayList<Object> zbytek = new ArrayList<Object>();
    for (int i = 0; i < obj.getCount(); i++) {
        mapI++;
        pracovniEle.put("I" + pl.getName() + "G" + target.getNamed().getName()
            + "O" + mapI, target.getHasObject().get(i));
    }
    for (int i = obj.getCount(); i < target.getHasObject().size(); i++) {
        zbytek.add(target.getHasObject().get(i));
    }
    updateGroup(target, zbytek);
}
```

Výpis 10: Změna počtu objektů simulací

Během upravování simulace výstupních objektů se projevila nepříjemnost, že musíme nějakým způsobem tyto objekty rozdělit mezi výstupní místa. Zároveň je v zadání požadavek, aby uživatel měl možnost zvolit, zda z přechodu bude vystupovat stávající objekt, nebo nový. Jako řešení celé této problematiky se nabízelo plánované využití skriptů. Z toho důvodu se elementy ukládají do hashmapy `pracovniEle`, kde klíč říká, zda objekt do přechodu vstupuje (`I`), nebo z něj vystupuje (`O`), z jakého a do jakého místa objekt jde, jaký je jeho typ(určený `GroupName`) a kolikátý v pořadí pro svůj typ a místo byl vybrán. Poté algoritmus prochází všechny výstupní hrany a dle násobnosti vytvoří nové objekty a také je přidá do mapy. Na tento kód se můžeme podívat ve výpisu 27 umístěném v příloze.

Uživatel poté bude moci ve scriptu určit, jestli jako výstupní objekt bude do místa předán nově vytvořený, nebo některý ze vstupních objektů. Pro ulehčení práce s objekty ve scriptu byla vytvořena třída `PesHelper`, která do proměnné `script` přidá komentáře s označením objektu a jeho popisem.

6.2 Řešení náhodné volby parametrů

Další požadavek zadání byl, aby simulace umožňovala náhodnou volbu zvolených parametrů. K řešení tohoto problému byla zvolena možnost definovat přechodu scénáře. Každý scénář má nastavený název, procentuální šanci, že se provede, minimální a maximální hodnotu a cenu. Právě různé scénáře zvětšují vyjadřovací schopnosti této Petriho sítě. To je zajištěno tím, že výstupním hranám by mělo být možné nastavit různé scénáře a tím zajistit, že by se provedly jen některé. První věcí, kterou bylo u scénářů potřeba vyřešit bylo špatné zadání pravděpodobnosti. Program totiž nijak nekontroluje, jestli součet pravděpodobností všech scénářů je 100%. Nejpříhodnější možností ošetření tohoto nedostatku byla kontrola a případná změna hodnot automaticky po uzavření dialogového okna přechodu. Tato ošetření bylo proto přidáno do metody `setTransitionImpl` ve třídě `TransitionAttributes`. Právě tato metoda je volána pro nastavení výstupu z dialogového okna. Jak tato oprava funguje můžeme vidět na výpisu 11.

```
EList<Scenario> scenare = transition.getHasScenarios();
int celkem = 0;
for (Scenario scen : scenare) {
    celkem += scen.getPercent();
}
if (celkem != 100) {
    float konst = (float) (100.0 / celkem);
    for (Scenario scen : scenare) {
        scen.setPercent(Math.round(scen.getPercent() * konst));

    }
}
```

Výpis 11: Změna počtu objektů simulací

Zde vidíme, že nejprve je spočten celkový součet procent u všech scénářů a pokud je jiný než 100, je spočtena konstanta, kterou jsou poté jednotlivé šance scénářů násobeny, a tím se jejich součet opraví na 100%. V tomto kroku je problém, že procentuální šance jsou ukládány jako `int`, a tudíž se může stát, že když zaokrouhlíme opravenou hodnotu typu `float`, získáme zase součet jiný než 100. Kupříkladu, když budeme mít 2 scénáře a dáme jim šance 1 a 7, pak po opravě se tyto šance opraví na 12,5 a 87,5. Po zaokrouhlení tady získáme šance 13 a 88, což nám v součtu dá celkem 101%. K odstranění této chyby bylo třeba provést další opravu modelu. V tom byl změněn typ z `int` na `float`. Po přegenerování přišla řada na opravu třídy `TransitionDialog`, kde bylo potřeba v metodě `getScenarios` upravit přetypování scénáře. Zároveň hned bylo přidáno ošetření chyb vznikajících při nevyplněné hodnotě a zachycení `NumberFormatException` vznikající při špatně zadané hodnotě.

S naplněnými hodnotami mohla začít simulace. V té bylo potřeba v prováděném místě náhodně vybrat jeden přechod dle pravděpodobnosti. To bylo vyřešeno tak, že jednotlivým scénářům byly přiřazeny intervaly dle jejich velikostí, na což se můžeme podívat v tabulce.

Pravděpodobnost	Interval
12,5	$\langle 0; 12.5)$
12,5	$\langle 12.5; 25)$
25	$\langle 25; 50)$
50	$\langle 50; 100)$

Při sloučení těchto intervalů jsme získali spojitý interval $\langle 0; 100)$. Proto stačilo jen vygenerovat číslo z tohoto intervalu. K tomuto účelu je dostačující použití náhodně vygenerované floatové hodnoty pomocí metody `nextFloat` třídy `Random`. Zmiňovaná metoda vrací náhodné číslo z intervalu $\langle 0; 1)$. Tato hodnota je vynásobena $\times 100$, čímž získáme hodnotu z cílového intervalu. Poté se postupně procházejí scénáře a sčítá se jejich pravděpodobnost, dokud nebude tento součet větší, než vygenerovaná hodnota, protože v takovém případě je horní hranice intervalu větší, než hodnota požadovaná. Prakticky to můžeme vidět ve výpisu 12.

```
private Scenario getScenarioToDo(Transition trans) {
  if (trans.getHasScenarios().size() == 0) {
    return null;
  }
  float volba = new Random().nextFloat()*100;
  float celkem = 0;
  for (Scenario scen : trans.getHasScenarios()) {
    celkem += scen.getPercent();
    if (celkem > volba)
      return scen;
  }
  return null;
}
```

Výpis 12: Metoda pro náhodnou volbu scénáře.

Následně bylo nutné přistoupit v metodě `proved` k upravení algoritmu. Především bylo třeba změnit část řešící provádění výstupních hran a to tak, že algoritmus nejprve otestuje, zda metoda vrátila nějaký náhodný scénář, nebo je její návratová hodnota `null`. To by znamenalo, že přechod nemá definované žádné scénáře, a proto se provedou všechny výstupní hrany. Pokud by byl vrácen nějaký scénář, pak se u každé výstupní hrany testují scénáře, a pokud přechod obsahuje prováděný scénář, hrana se provede. Tuto část algoritmu můžeme vidět na výpisu 13.

```
boolean provest = false;
if (scenar != null) {
  for (Scenario scen : edge.getScenarios()) {
    provest |= scen.equals(scenar);
  }
}
if ((scenar == null) || provest) {
  //algoritmus provedeni hrany
}
```

Výpis 13: Algoritmu testující proedení hrany.

6.3 Implementace skriptovacího enginu

Jelikož si vývojáři jazyku Java uvědomují výhody skriptovacích jazyků, mezi něž patří především

- **Pohodlí**: Většina skriptovacích jazyků je dynamicky typována. Můžeme většinou vytvářet nové proměnné bez nutnosti deklamace datového typu a jednu proměnnou můžeme použít pro uložení více datových typů. Zároveň skriptovací jazyky zvládají automaticky konverze mezi různými typy např. číslo 10 na řetězec "10"a zpět.
- **Rychlejší vývoj**: Můžeme se vyhnout cyklu editace - kompilace - běh, ale po editaci rovnou spouštíme běh.
- **Rozšíření aplikace**: Můžeme "ztvárnit" části aplikace - například konfigurací, business logiky/pravidel, výpočetních a matematických výrazů pro finanční aplikace.
- **Příkazový řádek pro zkoušení**: Pro ladění, nastavení času a jiných runtime vlastností. Většina aplikací má konfigurační nástroj prostřednictvím webového rozhraní. Ale administrátoři a vývojáři často preferují příkazový řádek. Místo vymýšlení vlastního skriptovací jazyku ad-hoc za tímto účelem, může být použit již vytvořený skriptovací jazyk.

Pro účely použití skriptovacího jazyku v jazyku Java se používá `Java™ Scripting API`. S tím je možné napsat přizpůsobitelné aplikace v jazyce Java a nechat jejich přizpůsobení pomocí skriptovacího jazyku až na koncovém uživateli. Vývojář ani nemusí zvolit skriptovací jazyk během vývoje, protože pokud použije `JSR-223 API`, pak uživatelé mohou použít libovolný JSR-223 kompatibilní skriptovací jazyk [19].

Funkcionalita Java skriptování je umístěna v balíčku `javax.script`. Jedná se o relativně malé, jednoduché API. Výchozím bodem API je třída `ScriptEngineManager`. Tato třída může vytvářet různé skriptovací enginy, které načte z `.jar` souborů popisující jednotlivé jazyky. Dále vytváří instanci třídy `ScriptEngine`, která poté interpretuje jednotlivé enginy. Nejjednodušší způsob, jak skriptovací API používat je:

1. Vytvoříme objekt `ScriptEngineManager`
2. Z objektu třídy `ScriptEngineManager` získáme objekt `ScriptEngine`
3. Pomocí metody `eval` z objektu `ScriptEngine` vyhodnotíme skript.

Z důvodu komplikovanosti vývoje rovnou ve vyvíjené aplikaci pro simulaci mapy probíhal počáteční vývoj části řešící skriptování v samostatném projektu. Hlavní výhoda byla rychlost spuštění aplikace, při němž odpadlo načítání nového okna prostředí Eclipse. To značně ulehčilo vývoj. Při zkopírování ukázkového kódu z prvního příkladu příručky (výpis 14) bylo překvapivé, že kód hned fungoval a jeho integrace do aplikace se nezdálo být až tak složité.

Při pokusu změnit skriptovací engine z "JavaScript" na "groovy" dle příručky [21] vyvstal problém chybějící knihovny groovy-engine.jar. V příručce radí tuto knihovnu stáhnout na stránce https://scripting.dev.java.net. Zde je problém, že zmíněná stránka již neexistuje, a proto bylo třeba potřebnou knihovnu najít jinde. Při hledání bylo zjištěno, že původní stránka byla přesunuta na novou adresu. Tyto stránky ale vypadaly hotové jen z poloviny. Nakonec se v SVN na této stránce podařilo nějaké knihovny nalézt, ale byla zde knihovna groovy-all. Ta se později ukázala být taky potřebná, ale stále program bez třídy enginu hlásil chybu při pokusu o instancializaci Groovy enginu. Při dalším hledání se podařilo potřebnou knihovnu stáhnout ze stránek http://www.java2s.com/. Po přidání těchto dvou knihoven byl kód z výpisu 14 s novým enginem spustitelný bez chyb.

```
import javax.script.*;
public class EvalScript {
    public static void main(String[] args) throws Exception {
        // create a script engine manager
        ScriptEngineManager factory = new ScriptEngineManager();
        // create a JavaScript engine
        ScriptEngine engine = factory.getEngineByName("JavaScript");
        // evaluate JavaScript code from String
        engine.eval("print ('Hello, World')");
    }
}
```

Výpis 14: Příklad provedení skriptu.

Během dalších úprav kódu se ukázalo být velkou výhodou to, že skript s vloženou hashmapou pracuje jako s referenčním datovým typem, a proto se změny provedené skriptem ihned promítnou objektu původního programu, nad kterým skript prováděl změny. Příkladně to můžeme vidět na kódu 15. Na tom je vidět, že nejprve je třeba předat do skriptu proměnnou. Bez tohoto přidání by o ni skript nevěděl a nemohl s ní pracovat. V následujících dvou příkazech je nejprve změněn záznam v mapě, který vytvořil Java kód a druhým příkazem je do mapy přidá nový záznam.

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");

HashMap<String, Object> hm = new HashMap<String, Object>();
hm.put("hodn", 200);
engine.put("hm", hm);              // vložení proměnné do skriptu
System.out.println(hm);            // vypíše {hodn=200}
engine.eval("hm.hodn = 1;");       // změna hodnoty skriptem
engine.eval("hm.skrip = 8;");      // vlozeni hodnoty skriptem
System.out.println(hm);            // vypíše {hodn=1, skrip=8}
```

Výpis 15: Práce scriptu s mapou.

Po těchto zjištěních byl engine přidán do metody proved v simulaci a do referencí přidány potřebné knihovny. Při zkušebním spuštění, kdy bylo voláno provedení skriptu z výpisu 14, se bez chyb vypisoval požadovaný kód, čímž bylo potvrzeno, že engine funguje správně. Proto došlo odstranění tohoto zbytečného kódu, který byl nahrazen za-

voláním metody `eval` nad skriptem získaným od uživatele skrze dialogové okno. Před tímto voláním bylo však třeba nejprve enginu vložit všechny objekty, které do místa vstupují a z něj vystupují. Jak tyto objekty získat již bylo vyřešeno v kapitole 7.1. Nyní proto stačilo jen iterovat mapu `pracovniEle` a do scriptu vkládat záznamy, kdy klíč záznamu udával, pod jakým jménem bude objekt uložený v hodnotě do skriptu vložen.

V tomto stádiu program umožňoval nastavit výstupním objektům uživatelské atributy. Stále nebylo možné zajistit, zda se jako výstupní objekty použijí již existující ze vstupu, nebo se vytvoří nové. Tuto funkcionalitu jde rovněž řešit skriptem tak, že jako výstupní objekt nastavíme některý ze výstupních, jako to je na výpisu 16.

```
// gen Ip1GaO1 – objekt typu a z mista p1.
// gen Op2GaO1 – objekt typu a do mista p2.

Op2GaO1 = Ip1GaO1;
```

Výpis 16: Ukázka nastavení vstupního objektu jako výstupní.

Toto přiřazení se ale neprojeví na vytvořených objektech a je proto nutné po provedení skriptu z enginu objekty načíst zpátky. Toho je možné docílit metodou `engine.get`, které je v parametru předán název objektu, jež chceme načíst. To zajistilo možnost definovat přechodu, zda pracuje se stávajícími objekty nebo vytváří nové.

Ještě bylo třeba ošetřit chyby, které program hlásil z důvodu chybně napsaného, nebo prázdného skriptu. K tomuto účelu bylo třeba nejprve vymazat komentáře. Pokud skript bez komentářů nebyl prázdný, pak se teprve algoritmus pokusil provést skript, a při neúspěchu odchytit vzniklou výjimku.

6.4 Doplnění chybějících částí

Jednou z chybějící částí simulace bylo dořešení kapacity počtu objektů v místě. K vyřešení tohoto nedostatku bylo nutné upravit metodu `isProveditelny`. V této metodě bylo doposud pouze porovnáváno, jestli je ve vstupních místech přechodu dost objektů. Stačilo tedy tento kód zkopírovat, zaměnit list vstupních míst za list výstupních a změnit podmínku. Podmínka se změnila z původní testující, jestli je počet objektů v místě větší, nebo roven násobnosti hran na podmínku novou, které testuje, zda je počet objektů v místě plus násobnost hrany menší, či rovna kapacitě místa pro daný typ objektu. Neomezenou kapacitu místa lze nastavit nastavením kapacity na `0`. Na první pohled to vypadá, že tím jsme přišli o možnost nastavit místu restrikci znemožňující vložení objektu, ale to již máme ošetřené tím, že do místa není možné vložit objekt, jehož typ není zahrnut v listu kapacit. Celé to můžeme vidět v příloze ve výpisu 28.

Následující úpravou byla další změna modelu, kdy došlo ke změně datových typů některých proměnných a přidání několika metod. Jmenovitě šlo o změnu proměnných `min`,`max` a `cost` ve třídě `Scenario`, které byly přetypovány ze `String` na `int`. Poté byl přidán objekt `EOperation` s názvem `getValue` a datovým typem `int`. Tento objekt po vygenerování kódu vytvoří ve třídě metodu s tímto názvem a typem. Když bylo toto hotovo, přišla na řadu třída `Transition`, kde byly stejným způsobem přetypovány proměnné `timeMin`, `timeMax`, `countMin`, `countMax` a přidány metody `getTime`, `getCount`.

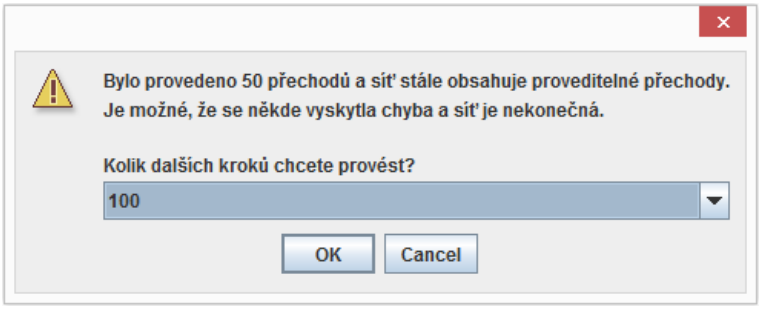
Po této změně došlo k novému vygenerování kódu. Po opravách datových typů při načítání scénáře v dialogovém okně ještě bylo třeba doplnit kód k nově vygenerovaným metodám. Účelem těchto metod je vrátit náhodnou hodnotu z intervalu $\langle min; max \rangle$. Aby toto fungovalo jak má, bylo potřeba ještě upravit `set` metody maximálních hodnot tak, že pokud je maximální hodnota zadaná uživatelem menší než hodnota minimální, pak tuto minimální zároveň nastaví i jako maximální. Tato podmínka je znovu testována i v metodě, kdy pokud zjistí minimum větší než maximum, pak vrací minimum, jinak vrátí náhodný prvek z výše zmíněného intervalu. To si můžeme prohlédnout na výpisu 17, kde je kód pro `getValue` třídy `Scenario`. Pro zbylé dvě nové metody je algoritmus stejný, jen pracuje s jinými proměnnými.

```
public int getValue() {
    if (min>max)
        return min;
    return new Random().nextInt(max − min + 1) + min;
}
```

Výpis 17: Vyběr hodnoty scénáře.

6.5 Přidání dalších akcí

Simulace Petriho sítě umožňující provádění po jednotlivých krocích je pro účel ukázky funkčnosti dobrá, ale chceme-li provést nějakou větší síť pro získání výsledku, pak bychom se uklikali. Z toho důvody byla přidána akce provádějící více přechodů. Po přidání položky v menu a akce, což bylo pouze zkopírování a přejmenování stávající akce, bylo potřeba v simulaci vytvořit metodu, která by byla touto novou akcí volána. Na vytvoření této metody nebylo nic složitého. Stačilo pouze v cyklu zjišťovat, zda se může něco provést a poté náhodně některý z přechodů provádět. Tady bylo potřeba ošetřit situaci, kdy je prováděná síť nekonečná[4]). Z toho důvodu byl příkaz omezen na provedení padesáti přechodů.



Obrázek 10: Potvrzení více přechodů

[4]Jedná se o Petriho síť bez uzamčení(viz. def. 2.11)

Po jejich provedení vyskočí dialogové okno oznamující, kolik cyklů již bylo provedeno a upozorňující na možnou nekonečnost sítě. Zároveň se ptá uživatele, kolik dalších přechodů chce provést. Zde si uživatel vybere z možností 10, 50, 100, 500, 1000 a po kliknutí na ok se provede vybraný počet průchodů cyklu a poté se znovu objeví dialogové okno. Toto okno můžeme vidět na obrázku 10 a kód metody v příloze ve výpisu 29.

Další pro nadcházející práci potřebnou akcí je možnost vrátit síť do původního stavu. To bude potřeba především kvůli opakovanému spouštění. K tomuto účelu byla do menu přidána další položka. K nové položce menu musela být vytvořena příslušná akce, ke které bylo třeba vytvořit metodu, kterou bude v metodě `run` volat.

Abychom věděli, jak vypadala původní síť k níž se chceme vracet, bylo potřeba vytvořit si její kopii. K tomu je přímo v EMF funkce `copy`, která po zavolání vytvoří kopii objektu, který je jí předán v parametru. V našem případě chceme tvořit kopii objektu procesu, protože ten v sobě uchovává celou síť. Z důvodu špatně vytvořeného předávání odkazu na síť při provedení validace po každém uložení je však třeba zajistit, aby se již vytvořená první kopie nepřepisovala. Toho docílíme jednoduše tak, že kopii budeme vytvářet pouze v případě, že proměnná, do které ji ukládáme, ještě nebyla vytvořená. Celou tuto kontrolu a následné vytváření kopie můžeme vidět ve výpisu 18.

```
if (procOld == null) {
    procOld = EcoreUtil.copy(proc);
}
```

Výpis 18: Vytváření kopie hlavního procesu

Když máme vytvořenou kopii, může být napsána metoda vracející stav sítě do původního stavu. K tomu by mělo stačit u vykresleného procesu přepsat hodnoty listů `hasElement` a `hasGroupName`. To zajistí provedení dvou requestů, které vidíme na výpisu 19. Ty jsou provedeny po zavolání akce a po jejich dokončení je mapa obnovena do původního stavu. Bohužel je zároveň provedená nepříjemná akce, kdy se mapa automaticky urovná na jednu přímku.

```
mojeAkce.run(proc, ModelBPPackage.eINSTANCE.getProcess_HasElement(),procOld.
    getHasElement());
mojeAkce.run(proc, ModelBPPackage.eINSTANCE.getProcess_HasGroupName(),procOld.
    getHasGroupName());
```

Výpis 19: Obnovení stavu sítě z uloženého objektu

Z toho důvodu bylo potřeba hledat jinou alternativu, jak síť obnovit. Jako příhodné řešení se jevilo obnovit pouze počty a stav objektů v přechodech. Tento zdánlivě snadný úkol se ale ukázal jako složitý problém. Na první pohled se zdálo, že bude stačit pouze projít pole všech objektů, vybrat z nich jen místa a ke každému místu najít jeho kopii a poté jednoduše list objektů původního místa přepsat listem z kopie, jak to můžeme vidět na výpisu 20.

```
for (Element ele : proc.getHasElement()) {
    if (ele instanceof Place) {
        for (Element eleOld : procOld.getHasElement()) {
            if (eleOld instanceof Place) {
                if (eleOld.getName().equals(ele.getName())) {
                        mojeAkce.run(ele, ModelBPPackage.eINSTANCE.getPlace_HasObject(), eleOld.
                            getHasObject());
                }
            }
        }
    }
}
```

Výpis 20: Obnovení objektů v místě

Když byl tento kód proveden, vykreslená síť se na obrazovce vrátila do původní podoby, ale nebylo již možné provést žádný přechod, či síť uložit. Zde vyskočila chybová hláška velmi podobná té na obrázku 9, která oznamovala, že skupiny objektů v nových listech mají přiřazený typ (daný instancí třídy `GroupName`), který není uložen v žádném objektu. Tento problém vznikl tím, že když se vytváří kopie sítě, zároveň je vytvořena kopie těchto objektů. Když je poté síť z kopie obnovena, obnovené skupiny mají za typ nastavené právě tyto kopie, které ovšem původní síť nezná. Z tohoto důvodu je potřeba těmto objektům před jejich nahráním přenastavit typ. K tomu slouží algoritmus na výpisu 21, který nahradil samotnou akci v předchozím výpisu 20.

```
if (eleOld.getName().equals(ele.getName())) {
    mojeAkce.run(ele, ModelBPPackage.eINSTANCE.getPlace_HasObject(), ((Place) eleOld).
        getHasObject());
    for (Group group : ((Place) ele).getHasObject()) {
        for (GroupName groupName : proc.getHasGroupName()) {
            if (groupName.getName().equals(group.getNamed().getName())
            && groupName.isActive() == group.getNamed().isActive()) {
            mojeAkce.run(group, ModelBPPackage.eINSTANCE.getGroup_Named(), groupName);
            break;
            }
        }
    }
}
}
```

Výpis 21: Oprava typu objektů před aktualizací

Při tvorbě tohoto kódu zabralo nejvíce času hledání důvodu vzniku `nullPointerException` výjimky, která vznikala v jeho první verzi, když se první skupinám nastavoval nový typ a až poté byly tyto skupiny přidány do místa. Dle chybového výpisu vše vznikalo při volání akce na změnu proměnné `group`. Jelikož byl výpis hodně chaotický, chyba vyvstala až ve vyšší vrstvě celého programu, zdálo se, že problém nevzniká v algoritmu. Zvláště i když při debugování bylo zjištěno, že žádná z hodnot neměla hodnotu `null`. Po změně pořadí volání příkazů, aby se nejprve přehrál list skupin a poté se těmto skupinám nastaví správný typ, funguje algoritmus bez chyby.

Aby popsaný algoritmus fungoval správně musíme zajistit, že místa budou mít unikátní pojmenování a žádná 2 místa se nebudou jmenovat stejně. K zajištění této podmínky poslouží v kapitole 4.3 vytvořené validační pravidlo, jež kontroluje celou síť a prozatím slouží jen pro načtení instance hlavního procesu. Pomocí debugování a procházení stromu volání bylo nalezeno místo, kde se načítá síť z uloženého dokumentu. Konkrétně se jedná o metodu `setDocumentContent` ve třídě `BPmodelDocumentProvider`. Do tohoto místa bylo z validačního pravidla přemístěno uložení instance hlavního procesu. Tím jsme měli jedno nevyužité validační pravidlo. Do něj byl přidán onen algoritmus kontrolující unikátnost názvů. Jelikož se ukázalo, že i když algoritmus vyhodí výjimku, která se nikde nezobrazí, bylo ještě nutné změnit typ validovaného objektu. Tato změna se provádí v souboru `BPmodel.gmfmap`, kde byl změně typ validované položky z procesu na místo. Rovněž byla hned změněna chybová hláška. Po této změně a novém vygenerování již validace fungovala a při uložení označila místa s duplicitním jménem. To ovšem nevyřešilo problém, který by vznikl, kdyby někdo zavolal akci pro obnovení sítě před uložením, kdy ještě neproběhla validace, a tudíž stále mohou být vytvořená dvě místa se stejným jménem. Aby se tomuto předešlo, byla validačnímu pravidlu nastavena možnost validovat v live modu. To zajistilo, že validace probíhá po nastavení metody a je-li jméno duplicitní, pak není nastaveno.

Výhodou a zároveň nevýhodou současné akce pro obnovení je, že vrací pouze stav objektů v místech, nikoli ostatní nastavení, jako je skript, násobnost hran, kapacita míst či scénáře. Dokonce ani nejsou obnoveny odstraněné, a vymazány přidané hrany, místa a přechody. Proto bylo třeba přidat další dvě akce.

- **Akce pro obnovení celé mapy** - pro případ, když uživatel změnil nějaké hodnoty a zjistil, že někde udělal chybu. K tomu využijeme již dříve popsané dva requesty, které jsme dříve viděli na výpisu 19.

- **Akce pro uložení aktuálního stavu** - když uživatel před začátkem simulace změní nastavení sítě a přeje si tyto změny již trvale uložit, například protože plánuje další změny, jejichž výsledek předem neví, a mohl by mít zájem obnovit síť do jím uloženého stavu. Toho je docíleno přenastavením proměnné `procOld` na kopii aktuálního procesu z proměnné `proc`. Při implementaci této funkčnosti vznikla další chyba, kdy po obnovení takto uloženého stavu byly původní objekty obnoveny, ale v místech se zdvojnásobil počet objektů. Celé to bylo způsobeno pořadím volání. Jak se během debugování chyby ukázalo, framework vytvářející kopie nejprve nastavil hodnotu `count`. Metoda pro její nastavení však v důsledku změny jejího kódu, jež byla popsána dříve v kapitole 5.4, zároveň do listu vytvoří objekty. Když potom framework kopíruje samotné objekty, list už nevyčistí a pouze přidá kopie objektů. Tento problém vyřešila změna pořadí proměnných v modelu. Po jeho změně a vygenerování problém zmizel a akce fungovala správně.

Při pozdějších opravách byl kód dále upraven tak, že z něj bylo odstraněno procházení cyklu původního procesu a porovnávání jmen, aby byla nalezena kopie aktuálně hledaného jména. Odstranění tohoto cyklu zajistilo vytvoření hashmapy uchovávající

dvojici původní objekt a jeho kopie. Tato hashmapa je naplněna bezprostředně po vytvoření kopie. K naplnění stačí projít elementy a přiřadit k sobě elementy na stejných pozicích, protože kopírování probíhá postupně a pozice jsou zachovány. Tento jednoduchý algoritmus je možno vidět ve výpisu 22.

```
HashMap<Element, Element> copyMap = new HashMap<Element, Element>();
procOld = EcoreUtil.copy(proc);
for (int i = 0; i < proc.getHasElement().size(); i++) {
    copyMap.put(proc.getHasElement().get(i), procOld.getHasElement().get(i));
}
```

Výpis 22: Vytvoření hashpapy mapující dvojice objektů původní-kopie

Tato mapa umožnila zjednodušit základní kód z výpisu 20 na současnou verzi na výpisu 20. Zde můžeme vidět, že byl vyhozen vnitřní cyklus, podmínka kontrolující typ objektů a podmínka porovnávající jména. Zároveň párování pomocí hashmapy obchází problém, který by vznikl v případě neunikátních jmen.

```
for (Element ele : proc.getHasElement()) {
    if (ele instanceof Place) {
        mojeAkce.run(ele, ModelBPPackage.eINSTANCE.getPlace_HasObject(),
        ((Place) copyMap.get(ele)).getHasObject());
        //kód opravující typ objektů
    }
}
```

Výpis 23: Obnovení objektů v místě

7 Dokončení

7.1 Rozšíření modelu a simulace o čas

Po dohodě s vedoucím diplomové práce bylo rozhodnuto odložit vytvoření grafu a statistik na úkor jiných věcí. Hlavní důvod byl ten, že požadovaný graf by měl být graf zobrazující historii jednotlivých objektů. Tady byl problém, že v síti doposud nebyl zahrnut časový aspekt a z toho důvodu by takto vytvořený graf mohl být matoucí. Bylo tedy rozhodnuto, že důležitější je rozšířit simulaci o časový aspekt, o jehož vytvoření není v zadání práce žádná zmínka. Jelikož se má jednat o zachycení životního cyklu objektů, byl zvolen přístup časovaných tokenů. To znamenalo změnit model a přidat do třídy `Object` proměnnou `time`. Tato proměnná, jak její název napovídá, reprezentuje v teorii (kapitola 2.5) zmíněnou časovou značku θ_t. Ta udává, v jakém čase je objekt možno nejdříve použít.

Po této úpravě modelu mohla začít úprava simulace. Po zanalyzování nového požadavku bylo zjištěno, že je nutné vytvořit globální hodiny a novou funkci pro získání proveditelných přechodů. S přidáním globálních hodin problém nebyl, protože to spočívalo pouze v tom, vytvořit ve třídě simulace lokální proměnnou. Nyní bylo třeba napsat novou metodu pro zjištění proveditelnosti a získání množiny přechodů proveditelných v aktuálně nastaveném globálním čase. Nutnost vytvořit samostatnou novou metodu zahrnující čas namísto pouhé úpravy stávající metody byla třeba z důvodu, že stávající metoda bude nadále potřeba. Důvod je jednoduchý. Může se stát, že nová metoda vrátí výsledek, že za stávajícího nastavení času není žádný přechod proveditelný. To ale může znamenat, že síť je mrtvá, nebo pouze za současného času není řádný přechod proveditelný, ale v budoucnu existuje proveditelný přechod. Samotné zjišťování poté probíhá tak, že je nejprve obecně zjištěno, zda je bez ohledu na aktuální čas nějaký přechod proveditelný a do proměnné uložena množina těchto přechodů. Pokud je množina prázdná, znamená to, že síť je v aktuálním značení mrtvá a uživateli je vypsána zpráva u ukončení simulace. V opačném případě je tato množina předána jako vstup metodě zjišťující aktuálně proveditelné přechody. Pomocí té získáme množinu aktuálně proveditelných přechodů. V případě, že je tato množina prázdná, pak díky předchozímu zjištění, že síť není mrtvá, víme, že v budoucnu je nějaký přechod proveditelný. Proto je globální čas navyšován do doby, dokud nebude některý přechod v novém čase proveditelný.

K posouzení proveditelnosti v čase slouží metoda `isAktualneProveditelny`, která jako vstupní parametr obdrží přechod k otestování. Zde na rozdíl od obecného testování nestačí pouze zjistit, zda ve skupině, ze které jsou objekty brány, je dostatečný počet objektů, ale jestli mezi těmito objekty je dostatečný počet aktuálně použitelných objektů (tj. takových, které mají `time` $\leq globalniCas$). Z toho důvodu je třeba nejprve projít cyklem všechny objekty a spočítat ty, které vyhovují podmínce. Poté je porovnán tento počet s počtem požadovaným hranou.

Úprava metody `proved` řešící samotné provedení již spočívala pouze v určení doby trvání přechodu a na konci metody nastavení nového času výstupním objektům. K určení doby, jakou bude přechod trvat, slouží hodnoty `min`, `max` u scénáře. Při provádění se náhodně zvolí scénář a z něj je náhodně zvolena hodnota z intervalu $\langle min, max \rangle$. V

případě, že přechod nemá nastavené žádné scénáře, je interval získán přímo z přechodu, kde je tyto hodnoty nutné nastavit v okně properties.

Následovalo vyřešit, jak ve vykreslené mapě zobrazit toto omezení, kdy je kupříkladu v místě ve skupině 10 objektů, ale z nich jen 6 je aktuálně proveditelných. To bylo zprvu vytvořeno tak, že v lokální metodě vykonávající akci měnící počet objektů byl u měněné skupiny přenastaven titulek kupříkladu na hodnotu `6 (10)`, což znamená, že ve skupině je 10 objektů a z nich 6 je možné použít.

Tato metoda vykreslení se při zkoušce běžícího programu ukázala být chybnou. Problém se projevil při pokusu o uložení, kdy program neuměl uložit výše popsaný text jako integer. Za tímto účelem bylo třeba přesunout logiku tohoto vykreslení do samotného modelu. Zde byla vytvořena, metoda, která měla k tomuto účelu sloužit, ale tu nebylo možné v souboru `BPmodel.gmfmap` spárovat se zobrazovanou skupinou. Z toho důvodu bylo nutné místo metody vytvořit derivovanou[5] unsettable proměnou, následně bylo tělo `get` metody této proměnné nahrazeno tělem nahrazené metody. Z důvodu změny zmíněného souboru bylo třeba nově vygenerovat kód diagramu. Zároveň vyvstal problém, že z projektu, ve kterém je definován model, není nijak možné získat přístup k instanci simulace uchovávající nastavený globální čas. Z toho důvodu byl globální čas přesunut ze simulace do instance hlavního procesu.

Během ladění maličkostí byla objevena závažná chyba v metodě `proved`. Problém spočíval v opomenutí kontroly použitých objektů, takže i když byl některý objekt v aktuálním čase nepoužitelný, program jej používal. Za účelem odstranění tohoto nedostatku byla ve třídě `Group` přidána metoda `getHasEnabledObject` vracející pouze použitelné objekty, které mají nastavený čas menší, než je čas globální. Tato metoda byla nově použita při zkoušení aktuální proveditelnosti. V metodě `proved` bylo nutné oproti očekávání změnit více, než pouze metodu získávající objekty. Doposud získání objektů ze skupiny probíhalo tak, že probíhal cyklus od 0 do počtu objektů daných násobností hrany a ten vybral objekty na těchto pozicích jako pracovní. Poté probíhal druhý cyklus od počtu objektů daných násobností do celkového počtu objektu ve skupině a ten přidával tyto objekty do listu, který byl nastaven místu jako zbylé objekty. Zde není možné zajistit, aby se pracovní objekty braly z listu aktuálně proveditelných a zbytek vytvořil ze všech. Proto musel být vytvořen nový způsob vybírání objektů a získání zbytku nepoužitých. Tento nový způsob spočívá v tom, že nejprve jsou všechny objekty ze skupiny uloženy do listu a poté jsou teprve vybírány pracovní objekty z listu vráceného metodou `getHasEnabledObject`. Při vybrání je objekt uložen do pracovní hashmapy a metodou

[5] derivovaná proměnná - proměnná, jejíž hodnota není nikde uložena, ale získána z hodnot jiných proměnných

`remove` vymazán z listu všech objektů. Tento kód si můžeme prohlédnout na výpisu 24.

```
List<Object> zbytek = new ArrayList<Object>();
for (Object o : target.getHasObject()) {
    zbytek.add(o);
}
for (int i = 0; i < obj.getCount(); i++) {
    mapI++;

    pracovniEle.put("I" + pl.getName() + "G"
            + target.getNamed().getName() + "O" + mapI, target
            .getHasEnabledObject().get(i));
    zbytek.remove(target.getHasEnabledObject().get(i));

}
updateGroup(target, zbytek);
```

Výpis 24: Nový způsob získání pracovních objektu a zbytku skupiny.

7.2 Zachycení záznamu běhu

Pro budoucí práci na programu je třeba zaznamenat průběh simulace a historii použití objektů. Z toho důvodu vznikl balíček `zaznam`. V něm je umístěna třída `PesLogger`. Tato třída uchovává záznamy pro jednotlivé simulace. Jsou zde ukládány dva logy.

- Obecný log - uchovává, jaký přechod a scénář proběhl. K tomu je uloženo, jak dlouho provedení trvalo. Nad tímto logem je možné pomoci statistických metod jednoduše vypočíst pravděpodobnost, s jakou který přechod probíhá, jakou pravděpodobnost mají jednotlivé scénáře u přechodu a jaká je průměrná doba provedení přechodu a scénáře.
- Historie objektů - v tomto logu je zachycena historie jednotlivých objektů sítě. Je zde uloženo v jakém čase, jakým přechodem a dle jakého scénáře byl objekt použit, a jak dlouho toto použití zabralo. Tento záznam poslouží k tomu, když simulujeme kupříkladu výrobní proces a chceme modelovat grafem, jak a kdy byly jednotlivé objekty do procesu zapojeny.

Oba dva logy jsou ukládány do listu. Díky tomu jsou při opakovaní simulaci uloženy logy za všechny běhy a následně je můžeme vzájemně porovnat.

7.3 Doplnění guardu přechodu

Věcí, která byla během programování opomenuta, bylo vyhodnocení podmínky přechodu, tzv. guardu. Jedná se o podmínku, která na základě hodnot vstupních objektů jakoby vyhodnotí jejich validitu dle definovaných podmínek a dle toho přechod provede či neprovede. Z toho důvodu bylo nutné předělat provádění tak, aby se objekty ze skupin neodebíraly hned, ale aby se pouze vytvořil záznam změn k provedení a změny byly provedeny až po vyhodnocení guardu. To je zajištěno tím, že místo okamžitého volání

metod `updateGroup(Group, List<Object>)` a `updatePlaceAddGroup(Place, Group)` jsou jejich vstupní parametry ukládány do lokálních map. Tato změna rovněž umožnila nahradit strategii výběru objektů z dosavadního vybírání prvních ve frontě na výběr čistě náhodný. Poté je během přípravy proměnných nastavených ve skriptu přidána booleanovská proměnná `guard` defaultně nastavena na hodnotu `true` a po provedení skriptu je tato proměnná získána zpět a porovnána její hodnota. Je tudíž jen na uživateli, zda ve skriptu vstupní podmínku definuje a její výsledek uloží do proměnné `guard` či nikoli.

Zjistí-li program, že byla ve skriptu hodnota proměnné `guard` nastavena na `false`, pak okamžitě ukončí provádění přechodu s návratovou hodnotou `false`. V opačném případě program pokračuje provedením odložených operací, jejichž vstupy má uložené v mapách a dále provedením algoritmu získávajícího výstupní objekty ze skriptu a jejich vložení do výstupních míst. Po úspěšném provedení je uložen záznam o provedení a vrácena návratová hodnota `true`.

Jak je patrné, bylo nutné změnit návratový typ metody `proved` na `boolean`. Důvod pro tuto změnu je ten, že program neumí vyhodnotit `guard` během zjišťování proveditelnosti. Především proto, že k jeho vyhodnocení dochází až při provedení skriptu. Návratová hodnota by tudíž měla předejít situaci, kdy program označí přechod za proveditelný, ale v důsledku vyhodnocení guardu přechod nad aktuálními daty proveditelný není. Proto musíme zjistit, zda přechod byl proveden nebo ne. Tím předejdeme zacyklení, kdy se program snaží donekonečna provést přechod, který se v důsledku guardu ukončí a nedojde ani k žádné změně stavu.

7.4 Doplnění grafů a statistiky

Z důvodu požadavku na předělání sítě na časovanou byly poslední dva body minimalizovány tak, že byla pod tlačítko `Vypiš záznam` přidána akce, která do console vypíše minimální graf složený z _ a – zobrazující historii použití objektů, jak by mělo být znázorněno grafem. To znamená v jaké době byl který objekt používán. Jelikož nakonec předělání sítě na časovanou zabralo méně času, než bylo odhadováno, zbyl čas na vytvoření plnohodnotného grafu.

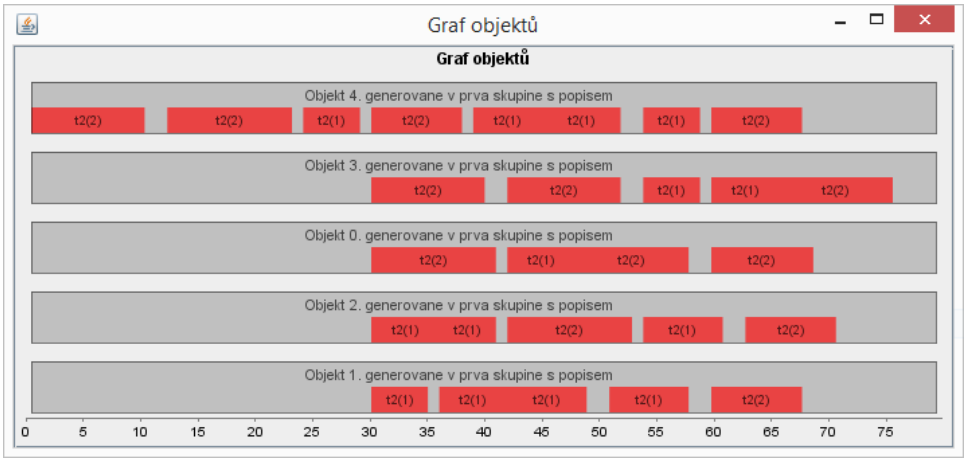
K vytvoření grafu posloužila knihovna `JFreeChart`. Jedná se o čistě Java knihovnu, vyvinutou pro snadné vytváření a zobrazení grafů v profesionální kvalitě. Knihovna je vydávána jako Open Source pod LGPL licencí, která dovoluje použití i v neotevřeném kódu. Ke knihovně je zdarma dostupná plná dokumentace, ale vývojářská příručka ve formátu PDF o více jak 750 stranách už je zpoplatněna nemalou částkou. Tato částka v sobě zahrnuje také roční aktualizace zdarma. To znamená, že v případě, když je vydána nová verze, dovolí uživateli stáhnout tuto novou verzi zdarma.

Při hledání návodu však posloužil návod [24], který se věnuje všemožným grafům vytvořeným v této knihovně a umožňuje stažení zdrojových kódů knihovny, které v sobě obsahují téměř 200 ukázkových příkladů použití různých grafů. Po procházení těchto návodů byl nalezen příklad, který ukazuje, jak vytvořit list samostatných grafů poskládaných po řádcích pod sebou. To je přesně ten graf, který potřebujeme. Nyní bylo ještě potřeba nalézt graf, který bude reprezentovat jednotlivé grafy v listu. Po pokusu o po-

užití několika kandidátů z řad kategorických[6] grafů bylo zjištěno, že k účelu vykreslení historie objektů se nejlépe hodí základní `XYPlot`, tedy graf, kterému jsou data k vykreslení předány pomocí bodů se souřadnicemi x,y.

Tento graf byl vložen na místo dosavadního vypisování `-,_` do console. Toto nahrazení vypadalo tak, že se v místě, kde byl volán příkaz pro vypisování pomlček značících, že je v této době objekt používán, byl vytvořen obdélník, začínající na souřadnicích $[z, 0]$, pokračující do $[z, 1]$. Tím byla vytvořena náběžná hrana. Následně se pokračovalo body $[k, 1]$ a $[k, 0]$. Tím získáme obdélník začínající na z značící globální čas, kdy začal být objekt používán, a končící na k, spočteným jako $z + dobaProvedeniPrechodu$. Tímto způsobem se vytvoří datová základna a ta je předána jako parametr metodě `createSubChart`. Tato metoda slouží k vytvoření grafu nad těmito daty. Jako další dva parametry jsou předány řetězec uvádějící název objektu určený proměnnou `description` a list provádění přechodů, který k jednotlivým obdélníkům vloží popisek, o jaký přechod a scénář se jedná. Při zkoušení se však náhodně vyskytovala chyba, když dva obdélníky navazovaly bezprostředně na sebe. Tuto chybu vyřešilo zapamatování si poslední hodnoty a kontrola následující. Když program zjistí, že objekty na sebe přímo navazují, souřadnice se nevracejí do hodnoty 0, aby se následně znovu nastavily na 0 a poté 1. Právě toto někdy způsobovalo, že nevznikla náběžná hrana a místo obdélníku byl vykreslen trojúhelník začínající v hodnotě 0.

Z jednotlivých takových grafů byl poté sestaven a vykreslen jejich list. U toho byl problém v proměnné délce. Tento problém vyřešilo vložení celého objektu do `JScroolPane`, což umožnilo vykreslení v okně předem definované velikosti a listem, který se do tohoto okna nevejde, je možno rolovat pomocí kolečka myši. Ve výsledku graf vypadá jako na obrázku 11.

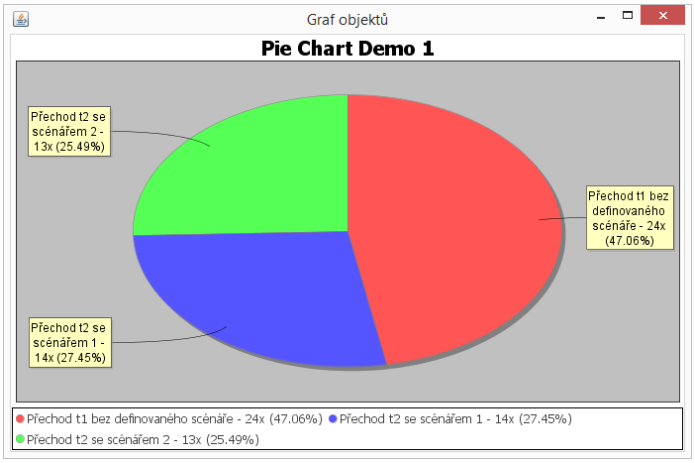


Obrázek 11: Graf použití objektů

[6]Grafu je zadáno pouze jednorozměrné pole hodnot a každý prvek je považován za jednu kategorii

Statistika

Z důvodu nedostatku času byla po dohodě statistika omezena pouze na vyhodnocení pravděpodobností, s jakou jsou spouštěné jednotlivé přechody a scénáře během celé simulace. Jako předešlý graf byla i tato statistika nejprve vytvořena pouze jako výpis do console, ale při vytváření grafu bylo i toto statistické porovnání převedeno do koláčového grafu, jaký je na obrázku 12.



Obrázek 12: Graf použití přechodů

7.5 Poslední drobné úpravy

Poslední provedenou úpravou bylo předělání menu, aby bylo přehlednější a uživatelsky přívětivější. Zároveň byla přidána položka, jež na základě uživatelem zadaného počtu opakování provede několik simulací. Dále byl do přílohy doplněn velice stručný návod, jak nastavit skript přechodu. Co se však nepodařilo nijak ošetřit je, že první kliknutí na jakýkoli z přidaných requestů ihned po spuštění neprovede žádnou akci. Akce jsou prováděny až po druhém kliknutí. Proto je vhodné před spuštěním automatické simulace nejprve kliknout na akci pro provedení náhodného přechodu.

7.6 Co chybí dodělat

V práci se stále nachází nesplněné úkoly, jejichž množství by vydalo na samostatnou práci. Jednak stále přetrvává nesplněný bod zadání původní práce na editoru, a to umožnění více kopií reprezentujících jeden objekt. Poté chybí vytvoření grafického zobrazení objektů za více běhů a statistické zhodnocení běhu sítě a dosažených výsledků za pomocí složitějších statistických metod. Mezi další věci, které by bylo potřeba dodělat patří změna editoru umožňující zobrazení listu objektů nacházejících se v určité skupině a možnost vidět jejich uživatelsky definovaných atributů. Toto je doposud možné pouze

otevřením souboru uchovávajícího model sítě. Rovněž by bylo vhodné oddělit vyhodnocování guardu do samostatného skriptu, aby jej bylo možné kontrolovat již během kontroly proveditelnosti, případně mít tak možnost otestovat podmínkou všechny objekty ze vstupního místa a vyřešit tak současný problém, kdy se může stát, že v místě je jeden objekt, který podmínku splňuje, a zbylé objekty, které ji nesplňují, a program zrovna tento splnitelný při náhodném výběru nevybere. Od implementace tohoto zkoušení všech možných variací bylo upuštěno z obavy, že zbytečné vykonávání celého skriptu pro každou permutaci by značně snížilo rychlost provádění simulace. V současné době je možné tento problém obejít pomocí vytvoření dvou výstupních míst a skriptem poté rozhodovat, do kterého místa objekt bude uložen.

8 Závěr

I přes nemalé problémy byla dle zadání doplněna simulace do již hotového editoru. Tato simulace umožňuje simulovat v editoru vytvořenou Petriho síť pracující s objekty podle pravidel definovaných v přechodu pomocí skriptu. Bohužel práce splnila bod zadání týkající se komplexní statistiky pouze povrchově. Rovněž se z důvodu již chybně vytvořeného editoru jeho autorem se nejedná o RCP aplikaci jako požaduje zadání, ale o Eclipse aplikaci, která ke svému spuštění vyžaduje plnohodnotné prostředí Eclipse.

Důvodem toho byla především koncepce rozdělení na dvě samostatné práce, které měly vzniknout souběžně. Autoři těchto prací měli spolupracovat. Jeden měl vytvořit simulaci a druhý tvořit editor, který by tuto simulaci zobrazoval a od uživatele načítal potřebné údaje. Bohužel vznikl pouze editor, který se navíc ukázal špatně analyzovaný a k úspěšnému dokončení simulace bylo potřeba jeho velkou část změnit. Tento úkol ale znamenal nejdříve znovu udělat velkou část práce a doplnit některé její chybějící body. Dle koncepce rozdělení práce mezi dva autory by tyto změny dělal autor editoru. Bez druhého autora ale tato práce navíc způsobila obrovské zdržení, jež zapříčinilo pouze částečné splnění posledních bodů zadání věnujících se statistice. Nicméně hlavní úkol vytvoření fungujícího simulátoru je splněn a v případě nutnosti provedení složitějšího statistického vyhodnocení, než pravděpodobnosti přechodů, je možné získat přístup k záznamu běhu, nad kterým je následně nutné statistické vyhodnocení provádět ručně.

9 Reference

[1] Pavel Kučera: *Editor Petriho sítí*, Diplomová práce VŠB 2012

[2] doc. RNDr. Jaroslav Markl: *Petriho sítě I*
Online dostupné z `http://drazdilova.cs.vsb.cz/Data/Sites/5/petrinet/petrinetsylabus.pdf`

[3] Prof. Ing. Ivo Vondrák, CSc.: *METODY BYZNYS MODELOVÁNÍ*, Ostrava 2004
Online dostupné z `http://vondrak.cs.vsb.cz/download/Metody_byznys_modelovani.pdf`

[4] Ing. Vladimír Janoušek: *Modelování objektů Petriho sítěmi* , VUTBR FIT 1998
Online dostupné z `http://www.fit.vutbr.cz/~janousek/publications/phdthesis.pdf`

[5] DARWIN, Ian F. *Java cookbook. 2nd ed.*, CA: O'Reilly, 2004, ISBN 05-960-0701-9. Online dostupné z: `http://it-ebooks.info/book/2249/`

[6] Scott Chacon: *Pro Git*. Vydal CZ.NIC, Praha, 2009. ISBN 978-80-904248-1-4.
Online dostupné z `http://www.root.cz/knihy/pro-git/`

[7] *Oficiální stránky platformy Java*
`http://java.com`

[8] *Úvod do jazyka Java*
`http://www.vogella.com/tutorials/JavaIntroduction/article.html`

[9] *Plánované milestones Java 8*
`http://openjdk.java.net/projects/jdk8/`

[10] *Oficiální stránka věnovaná představení Java 8*
`http://www.oracle.com/events/us/en/java8/index.html`

[11] *Úvod do Eclipse*
`https://www.eclipse.org/org/`

[12] *Oficiální stránky systému LaTeX*
`http://www.latex-project.org/`

[13] *David Martínek: LaTeXové speciality*
`http://www.fit.vutbr.cz/~martinek/latex/index.html`

[14] *Graphical Modeling Framework documentation*
`http://wiki.eclipse.org/Graphical_Modeling_Framework/Documentation`

[15] *Introducing the GMF Runtime*
`http://www.eclipse.org/articles/Article-Introducing-GMF/article.html`

[16] *Oficiální stránky Groovy projektu*
http://groovy.codehaus.org/

[17] *Groovy je žúžo*
http://www.java.cz/detail.do?articleId=8020

[18] *Záhada jménem Groovy*
http://blog.zvestov.cz/item/150

[19] *Oficiální dokumentace o scriptovaní*
http://docs.oracle.com/javase/7/docs/technotes/guides/scripting/programmer_

[20] *Projekt řešící knihovny pro skriptování v Javě*
https://java.net/projects/scripting

[21] *Způsob jak spouštět Groovy kód*
http://groovy.codehaus.org/JSR+223+Scripting+with+Groovy

[22] *Oficiální help stránka Eclipse*
http://help.eclipse.org/

[23] *Dokumentace třídy properties*
http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

[24] *Příklady použití knihovny JFreeChart*
http://www.java2s.com/Code/Java/Chart/CatalogChart.html

10 Seznam příloh

- **Příloha A** - Výpisy důležitých částí zdrojového kódu
- **Příloha B** - Příklady použití scriptů.
- **Příloha na CD/DVD**
  - text práce ve formátu PDF/A
  - složka `sources` obsahující zdrojové kody
  - složka `latextext` obsahující obrázky, pomocné knihovny a soubor `praceUTF.tex`, sloužící k vygenerování textu této diplomové práce
  - soubor `README.txt` obsahující návod spuštěni
  - soubor `eclipse.rar` obsahující zabalené prostředí, které nutné ke spuštění

A Vypisy důležitých částí zdrojového kódu

```
public class ChangeAction extends DiagramAction {

    EObject target = null;
    EStructuralFeature targetValue = null;
    Object newValue = null;

    protected ChangeAction(IWorkbenchPart part) {
        super(part);
    }

    public void run(EObject target, EStructuralFeature targetValue,
            Object newValue) {
        if (target != null && targetValue != null && newValue != null) {
            this.target = target;
            this.targetValue = targetValue;
            this.newValue = newValue;
            super.setTargetRequest(createTargetRequest());
            super.run();
        }
    }

    @Override
    protected Request createTargetRequest() {
        if (target == null || targetValue == null || newValue == null) {
            return null;
        }
        return new EditCommandRequestWrapper(new SetRequest(target,
                targetValue, newValue));
    }

    @Override
    protected boolean isSelectionListener() {
        return true;
    }

}
```

Výpis 25: Třída zastřešující provádění změn.

```
public static HashMap<String, Object> toMap(String text) {
    HashMap<String, Object> res = new HashMap<String, Object>();
    Properties props = new Properties();
    try {
        props.load(new StringReader(text.substring(1, text.length() - 1)
                .replace(",␣", "\n")));
    } catch (IOException e) {
        e.printStackTrace();
        return res;
    }
    for (Map.Entry<Object, Object> set : props.entrySet()) {
        char type = ((String) set.getValue()).charAt(0);
        String val = null;
        if (((String) set.getValue()).length() > 1)
            val = ((String) set.getValue()).substring(1);
        switch (type) {
            case 'S':
                res.put((String) set.getKey(), (String) val);
                break;
            case 'I':
                res.put((String) set.getKey(), Integer.parseInt((String) val));
                break;
            case 'D':
                res.put((String) set.getKey(), Double.parseDouble((String) val));
                break;
            case 'O':
                res.put((String) set.getKey(), val);
                break;
            default:
                break;
        }
    }
    return res;
    }
```

Výpis 26: Metoda pro převod řetězce na mapu

```
HashMap<String, ObjectPointer> pracovniEle = new HashMap<String, ObjectPointer>();

for (Edge edge : trans.getLinkedIn()) {
    HashMap<GroupName, Group> objMista = new HashMap<GroupName, Group>();
    Place pl = (Place) edge.getSource();
    for (Group obj : pl.getHasObject()) {
        objMista.put(obj.getNamed(), obj);
    }
    for (GroupPointer obj : edge. getMultiplicity ()) {

        int mapI = 0;
        Group target = objMista.get(obj.getNamed());
        ArrayList<Object> zbytek = new ArrayList<Object>();
        for (int i = 0; i < obj.getCount(); i++) {
            mapI++;
```

```
            ObjectPointer op = new ObjectPointer(target.getHasObject()
                    .get(i), target);
            pracovniEle.put("I" + pl.getName() + "G"
                    + target.getNamed().getName() + "O" + mapI, op);
        }
        for (int i = obj.getCount(); i < target.getHasObject().size(); i++) {
            zbytek.add(target.getHasObject().get(i));
        }
        updateGroup(target, zbytek);

    }
}
for (Edge edge : trans.getLinkedOut()) {
    HashMap<GroupName, Group> objMista = new HashMap<GroupName, Group>();
    Place pl = (Place) edge.getTarget();
    for (Group obj : pl.getHasObject()) {
        objMista.put(obj.getNamed(), obj);
    }
    for (GroupPointer obj : edge. getMultiplicity ()) {

        int mapI = 0;
        if (objMista.get(obj.getNamed()) == null) {
            Group nova;
            if (obj.getNamed().isActive()) {
                nova = ModelBPFactory.eINSTANCE.createActive();
            } else {
                nova = ModelBPFactory.eINSTANCE.createPassive();
            }
            nova.setNamed(obj.getNamed());
            objMista.put(nova.getNamed(), nova);
            updatePlaceAddGroup(pl, nova);
        }
        Group target = objMista.get(obj.getNamed());
        for (int i = 0; i < obj.getCount(); i++) {
            mapI++;
            Object o = ModelBPFactory.eINSTANCE.createObject();
            o.setDescription("gen");
            ObjectPointer op = new ObjectPointer(o, target);
            pracovniEle.put("O" + pl.getName() + "G"
                    + target.getNamed().getName() + "O" + mapI, op);
        }

    }
}
for (Entry<String, ObjectPointer> set : pracovniEle.entrySet()) {
    engine.put(set.getKey(), set.getValue().getObj());
}
```

Výpis 27: Simulace přesunu objektů mapu

```
for (Edge edge : trans.getLinkedOut()) {
    HashMap<GroupName, Integer> objMista = new HashMap<GroupName, Integer>();
    HashMap<GroupName, Integer> capa = new HashMap<GroupName, Integer>();
```

```
    Place pl = (Place) edge.getTarget();
    for (Group obj : pl.getHasObject()) {
        objMista.put(obj.getNamed(), obj.getCount());
    }
    for (GroupPointer obj : pl.getCapacity()) {
        capa.put(obj.getNamed(), obj.getCount());
    }
    for (GroupPointer obj : edge. getMultiplicity ()) {
        if (objMista.get(obj.getNamed()) == null
                || capa.get(obj.getNamed()) == null) {
            return false;
        } else {
            if (capa.get(obj.getNamed()) <= 0) {
                continue;
            } else {
                proveditelne &= (capa.get(obj.getNamed()) >= (objMista
                        .get(obj.getNamed()) + obj.getCount()));
            }
        }
    }
}
```

Výpis 28: Kód žešící omezení kapacity místa.

```
public void provedAll(ChangeAction action) {
    this.action = action;
    ArrayList<Transition> proveditelne = getProveditelne();
    Random rnd = new Random();
    int stop = 50;
    int p = 0;
    while (proveditelne.size() > 0) {
        stop--;
        p++;
        proved(proveditelne.get(rnd.nextInt(proveditelne.size())));
        proveditelne = getProveditelne();
        if (stop <= 0) {
            String[] possibilities = { "10", "50", "100", "500", "1000" };
            String s = (String) JOptionPane
                    .showInputDialog(
                            null,
                            "Bylo␣provedeno␣"
                                    + p
                                    + "␣přechodů␣a␣síť␣stále␣obsahuje␣proveditelné␣přechody.\nJe␣mož
                                        né,␣že␣se␣někde␣vyskytla␣chyba␣a␣síť␣je␣nekonečná.\n\nKolik␣
                                        dalších␣kroků␣chcete␣provést?␣",
                            "", JOptionPane.WARNING_MESSAGE, null,
                            possibilities , "100");

            if ((s != null) && (s.length() > 0)) {
                switch (s) {
                case "10":
                    stop += 10;
                    break;
```

```
                case "50":
                    stop += 50;
                    break;

                case "100":
                    stop += 100;
                    break;

                case "500":
                    stop += 500;
                    break;
                case "1000":
                    stop += 1000;
                    break;

                default:
                    break;
                }
            } else {
                return;

            }

        }
    }
    if (proveditelne.size() > 0) {
        JOptionPane.showMessageDialog(null,
                "Žádný přechod není možné provést", "",
                JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Výpis 29: Metoda pro provádění přechodů.

B Základy nastavení skriptu přechodu

Krom objektů jsou do skriptu vloženy tyto další proměnné. Těmi jsou

- `scen` - instance prováděného scénáře,
- `time` - čas, jak dlouho trvá provedení,
- `guard` - vstupní podmínka (viz. kapitola 7.3),
- `trans` - prováděný přechod.

Chceme-li používat guard, musíme jej nastavit dříve, než objekty!
Příklady použití scriptu:

- Jednoduché přesunutí objektu z místa a do místa b

```
//gen IaGGrO1 – objekt typu Gr z mista a.
//gen ObGGrO1 – objekt typu Gr do mista b.
ObGGrO1 = IaGGrO1
```

- Nastavení hodnoty uživatelského atributu objektu

```
//gen IaGGrO1 – objekt typu Gr z mista a.
//gen ObGGrO1 – objekt typu Gr do mista b.
ObGGrO1.prop.nazevAtributu = hodnota
```

  Pozor! Pokud předáváme na výstup vstupní objekt, je nutné nastavovat atributy tomuto vstupnímu objektu, nebo jen nejprve nastavit, jako v předchozím příkladě.

- Nastavení guardu jen na objekty s časem menším než 10.

```
//gen IaGGrO1 – objekt typu Gr z mista a.
//gen ObGGrO1 – objekt typu Gr do mista b.
guard = IaGGrO1.time<10
```

- Umístění vstupního objektu do místa dle podmínky

```
//gen IaGGrO1 – objekt typu Gr z mista a.
//gen Ob1GGrO1 – objekt typu Gr do mista b1.
//gen Ob2GGrO1 – objekt typu Gr do mista b2.
if ( podminka ) {
   Ob1GGrO1 = IaGGrO1
   Ob2GGrO1 = null
} else {
   Ob1GGrO1 = null
   Ob2GGrO1 = IaGGrO1
}
guard = IaGGrO1.time<10
```